ARM期末

23物联网工程 2023124306

第二章

2.1.1 冯·诺依曼结构与哈佛结构

冯·诺依曼结构和哈佛结构是两种不同的计算机体系结构，它们在处理数据和指令的方式上存在显著差异。

1. 冯·诺依曼结构（Von Neumann Architecture）

冯·诺依曼结构是由数学家约翰·冯·诺依曼于1945年提出的计算机体系结构，它是现代计算机的基础。冯·诺依曼结构的特点在于**指令存储和数据存储共用同一存储器和总线**，即程序代码和数据存储在同一内存中，并通过相同的总线进行访问。

特点：

- **单一内存空间**：程序代码和数据存储在同一个内存区域。
- **共享总线**：数据和指令的读取都需要通过同一条总线，导致数据访问的竞争。
- **顺序执行**：指令按顺序从内存加载到CPU进行执行。

优点：

- 设计简单，硬件实现较为容易。
- 可编程性强，可以在同一内存中存储程序和数据。

缺点：

- **瓶颈问题**：因为指令和数据共享同一总线，读取指令和数据时可能会发生冲突，造成性能瓶颈。
- **效率较低**：需要多次访问内存，导致执行速度受限。

2. 哈佛结构（Harvard Architecture）

哈佛结构由哈佛大学的研究人员在20世纪40年代提出，通常用于专用的数字信号处理器（DSP）和嵌入式系统中。它的特点是**将程序指令和数据存储分开**，指令和数据分别存储在不同的内存中，并通过不同的总线进行访问。

特点：

- **独立的指令和数据存储器**：程序代码和数据存储在不同的内存中。
- **独立的总线**：指令和数据通过独立的总线进行传输，避免了冯·诺依曼结构中的数据总线竞争问题。
- **并行处理**：能够同时访问指令和数据，提高了效率。

优点：

- **高效并行性**：由于指令和数据可以同时访问，执行效率大大提高。
- **减少瓶颈**：没有指令和数据竞争总线的问题，减少了访问延迟。

缺点：

- **硬件复杂**：需要更多的内存和总线，设计和实现较为复杂。
- **灵活性较差**：在内存空间分配上不如冯·诺依曼结构灵活，因为指令和数据是分开存储的。

冯·诺依曼结构与哈佛结构的比较：

特性	冯·诺依曼结构	哈佛结构
内存	程序和数据共用同一内存	程序和数据存储在不同的内存中
总线	使用单一总线传输数据和指令	使用不同的总线传输数据和指令
访问	指令和数据访问存在竞争	可以同时访问指令和数据
硬件复杂性	硬件实现简单	硬件实现较为复杂
执行效率	受限于总线瓶颈，效率较低	由于指令和数据分开存储，效率较高
应用场景	适用于一般计算机	适用于嵌入式系统、数字信号处理等

在嵌入式系统中的应用

在现代嵌入式系统中，ARM Cortex系列内核通常采用了**哈佛架构的变种**。ARM处理器采用哈佛架构的优点，尤其是指令和数据的分离，使得嵌入式系统在性能、功耗和实时性方面得到了很好的优化。尤其是Cortex-M3等内核，它们通过实现较高效的指令集（如Thumb-2）以及独立的存储和总线，提供了较好的运行效率。

2.1.2 ARM存储模式

ARM架构的存储模式主要决定了处理器如何访问内存中的数据。ARM体系结构采用了一种灵活的存储模式，使得内存访问高效且具有较好的性能。具体来说，ARM处理器支持字节（Byte）、半字（Halfword）、字（Word）和双字（Doubleword）的存储模式。

ARM存储模式的关键点：

1. **字节（Byte）**：最小存储单位为1字节（8位），ARM处理器可以读取和写入单个字节的数据。
2. **半字（Halfword）**：存储2字节（16位）的数据。ARM支持以半字为单位对内存进行访问。
3. **字（Word）**：存储4字节（32位）的数据。大多数ARM处理器支持4字节对齐的存储和访问。
4. **双字（Doubleword）**：存储8字节（64位）的数据。高端ARM处理器（如Cortex-A系列）可以支持双字存储和访问。

字节对齐

ARM处理器通常要求内存访问是对齐的，即访问的地址应当是相应存储单位的整数倍。例如，读取一个字时，地址必须是4的倍数。如果发生未对齐的访问，可能会导致性能下降或产生异常。

大端模式与小端模式

ARM架构支持两种字节序：大端模式（Big Endian）和小端模式（Little Endian）。大端模式是指数据的高位字节存储在低地址处，而小端模式则相反。ARM处理器可以根据需要在这两种字节序中切换。

2.1.3 CISC和RISC

CISC（复杂指令集计算机）和RISC（精简指令集计算机）是两种不同的指令集架构，它们在设计理念、指令集复杂性和性能上有很大区别。

1. CISC（复杂指令集计算机）

CISC架构的特点是具有**复杂的指令集**，每条指令通常能够执行多个低级操作，如内存读取、加法运算等。因此，CISC架构的指令数目相对较多，且每条指令的功能较为复杂。常见的CISC架构有x86架构。

特点：

- 每条指令可以执行多个操作，如内存访问和算术运算。
- 指令长度不固定，可以是1到多个字节。
- 提供复杂的寻址方式。
- 需要更复杂的解码器来解析指令。

优点：

- 较少的指令数目可以实现较为复杂的操作，节省存储空间。
- 提高了编程效率，因为通过一条指令就可以执行多个操作。

缺点：

- 指令集较为复杂，硬件实现难度较大。
- 指令的执行时间不一致，导致CPU的效率降低。

2. RISC（精简指令集计算机）

RISC架构的特点是具有**简化的指令集**，每条指令执行一个简单的操作，并且每条指令的长度通常是固定的。ARM架构就是典型的RISC架构，强调指令的简洁和执行效率。

特点：

- 每条指令执行一个简单的操作，通常为数据的加载、存储或算术运算。
- 指令长度固定，通常是4字节。
- 寻址方式简化，一般使用寄存器间接寻址。
- 解码和执行过程较简单，能够实现较高的指令执行速率。

优点：

- 硬件实现较为简单，设计和制造成本较低。
- 由于指令集较小，CPU的执行效率较高。
- 每条指令的执行时间相对固定，提高了流水线处理能力。

缺点：

- 需要更多的指令来完成复杂操作，相对于CISC可能需要更多的程序空间。
- 为了完成一个复杂的操作，可能需要多条指令。

CISC与RISC的比较：

特性	CISC架构	RISC架构
指令集复杂性	指令集复杂，支持多种操作	指令集简单，每条指令执行单一操作
指令长度	不固定	固定
执行时间	执行时间不一致	执行时间固定
硬件实现	硬件复杂，解码器较复杂	硬件相对简单，解码器简单
程序空间	通常需要较少的指令	需要更多的指令完成复杂任务
应用	适用于需要复杂操作的任务（如桌面计算机）	适用于高效、低功耗的嵌入式系统

ARM处理器采用了RISC架构，它的指令集小而高效，能够在较低的功耗下实现较高的性能，非常适合嵌入式系统和移动设备。

2.1.4 流水线技术

流水线技术是一种通过将指令的执行过程划分为多个阶段，并且让不同的指令在不同阶段并行处理，从而提高计算机系统执行效率的技术。

流水线的基本概念：

流水线技术将指令的执行过程划分为若干个独立的阶段（通常包括取指、解码、执行、访存和写回阶段），每个阶段完成特定的操作。处理器可以在同一时刻并行执行多条指令的不同阶段，从而提高整体指令吞吐量。

流水线的工作原理：

在没有流水线的情况下，CPU执行每条指令时会依次完成所有操作。采用流水线技术后，CPU会在同一时刻同时处理多条指令的不同阶段。例如，当指令1正在取指阶段时，指令2可以在解码阶段，指令3可以在执行阶段，这样就能够并行处理多条指令。

流水线的优点：

- **提高吞吐量**：由于多个指令并行执行，指令的执行速度得到了显著提高。
- **提高效率**：每个时钟周期都能完成一条指令的执行，减少了空闲时间。

流水线的挑战：

- **流水线停顿（Pipeline Stall）**：如果某条指令依赖于前一条指令的结果（如数据依赖），就会导致流水线停顿，影响性能。
- **分支预测**：当遇到条件分支指令时，处理器需要预测分支的结果，否则会造成流水线的清空（称为“流水线冲刷”），也会影响性能。

ARM Cortex-M3内核的流水线技术：

ARM Cortex-M3处理器采用了三级流水线技术，通常包括**取指阶段（IF）、解码阶段（ID）和执行阶段（EX）**。通过流水线技术，Cortex-M3能够在每个时钟周期内执行一条指令，极大提高了指令的处理效率。

2.2 ARM

2.2.1 ARM介绍

ARM（Advanced RISC Machines）是一种基于精简指令集（RISC）架构的处理器架构，最初由Acorn Computers公司于1980年代开发，并逐渐成为全球范围内广泛使用的微处理器架构之一。ARM架构以其高效、低功耗和良好的性能广泛应用于各种嵌入式系统、移动设备（如智能手机、平板电脑）、物联网设备以及一些高性能计算平台。

ARM处理器以其简洁的指令集和低功耗特性，满足了现代移动设备对计算性能和电池续航时间的高要求。ARM架构不直接制造芯片，而是授权其他公司（如高通、苹果、三星等）使用其架构来设计和生产具体的芯片产品。

2.2.2 ARM体系结构

ARM体系结构采用RISC（精简指令集计算）原理，意味着它使用的是数量相对较少且效率高的指令集。这种设计有助于提高处理器的执行速度和节省功耗，适合低功耗和高性能的需求。ARM架构的核心特点包括：

1. **精简指令集（RISC）**：ARM使用的指令集相较于CISC（复杂指令集计算）架构更加简单，指令集较少，但每条指令的执行效率较高，通常在一个时钟周期内完成。
2. **多核架构**：ARM处理器通常采用多核设计，支持并行处理能力，提升多任务和多线程的处理性能。现代ARM处理器普遍采用大核与小核结合的设计，以实现性能与功耗的平衡。
3. **低功耗设计**：ARM架构以其低功耗特性闻名，适合移动设备使用。ARM设计在节能方面采用了多种技术，如动态电压频率调整（DVFS）、动态功耗管理等。
4. **32位与64位架构**：ARM架构支持32位与64位两种模式。32位模式较早出现，广泛应用于嵌入式设备中；64位模式则在更高性能的设备中得到了应用，如高端智能手机、服务器等。
5. **高效的内存管理单元（MMU）**：ARM架构具有高效的内存管理能力，支持虚拟内存管理和地址转换，能在不同的运行环境下实现高效的内存访问。
6. **丰富的指令集扩展**：ARM架构支持多个扩展，包括NEON SIMD（单指令多数据）扩展用于多媒体处理，TrustZone用于安全性管理等。
7. **广泛的应用场景**：ARM处理器被广泛应用于从低功耗嵌入式设备到高性能服务器的各类领域。特别是在移动设备、物联网（IoT）、汽车电子、消费电子等领域有着显著的应用。

ARM架构的主要优势在于高效能与低功耗的结合，以及其灵活性，能够根据不同需求定制不同级别的处理器。ARM处理器的授权模式使得全球各地的芯片设计公司可以根据自己的需求进行定制，从而推动了ARM架构的广泛应用。

2.3 ARM Cortex-M3内核

ARM Cortex-M3内核是基于ARM的RISC架构设计的一款高效能、低功耗的微控制器内核，广泛应用于嵌入式系统、物联网设备以及实时控制应用。Cortex-M3内核主要特点是支持高度集成的功能、灵活的内存管理以及高效的中断处理。

2.3.1 内核架构

Cortex-M3内核采用的是**哈佛架构**的变种，指令和数据存储是分开的，采用独立的总线以提高内存访问效率。其内核架构的关键特性包括：

1. **处理器模式**：Cortex-M3处理器支持两种主要的执行模式：**Thread模式**和**Handler模式**。
   - **Thread模式**：这是处理器的正常运行模式，用于执行应用程序代码。
   - **Handler模式**：当发生中断或异常时，处理器切换到此模式执行中断处理程序。
2. **操作模式**：Cortex-M3支持两种主要的工作状态：**Privileged模式**（特权模式）和**Unprivileged模式**（非特权模式）。

    - **特权模式**：通常由操作系统或核心程序使用，具有更高的访问权限。
    - **非特权模式**：通常由应用程序使用，权限较低。
3. **指令集**：Cortex-M3采用**ARMv7-M架构**，支持**Thumb-2指令集**。该指令集结合了16位和32位指令，旨在提供较低的代码大小和较高的执行效率。
4. **多级流水线**：Cortex-M3使用三级流水线（取指、解码、执行），提高了指令执行的效率和吞吐量。
5. **内存保护单元（MPU）**：Cortex-M3内核内置了**内存保护单元（MPU）**，允许系统定义不同的内存区域及其访问权限，保护重要的数据和代码免受非法访问。
6. **系统控制单元（SCU）**：该单元处理系统相关的控制操作，如系统复位、时钟控制、硬件中断管理等。
7. **调试支持**：Cortex-M3内核支持多个调试接口，如JTAG、SWD（Serial Wire Debug）和CoreSight。它提供了执行跟踪、数据捕获和性能分析的功能，方便开发人员调试嵌入式应用。

2.3.2 寄存器

Cortex-M3内核中有多个不同类型的寄存器，用于存储执行过程中的数据、状态信息以及控制指令的执行。主要寄存器如下：

1. **通用寄存器**：Cortex-M3提供了16个通用寄存器（R0到R15），用于存储处理过程中使用的数据。主要的寄存器包括：
    - **R0-R12**：用于存储通用数据，供程序使用。
    - **R13（SP）**：堆栈指针寄存器，指向栈顶，栈是用于存储局部变量、函数返回地址等数据的内存区域。
    - **R14（LR）**：链接寄存器，存储函数调用的返回地址。
    - **R15（PC）**：程序计数器，存储下一条将要执行的指令地址。
2. **程序状态寄存器（xPSR）**：xPSR（Program Status Register）包含处理器状态信息，包括当前的程序执行状态、条件标志和异常信息。
    - **CPSR（Current Program Status Register）**：存储当前的处理器状态，如中断使能标志、当前处理模式等。
    - **IPSR（Interrupt Program Status Register）**：存储当前处理中断的ID。
    - **EPSR（Exception Program Status Register）**：在处理中断时使用，记录当前的异常状态。
3. **控制寄存器（CONTROL）**：CONTROL寄存器控制处理器的运行模式，决定当前执行的特权级别和堆栈指针的选择。
4. **中断控制寄存器（NVIC寄存器）**：Cortex-M3内核包含一组中断控制寄存器，用于管理和控制中断的优先级、使能和状态。

2.3.3 存储结构

Cortex-M3的存储结构包括程序存储器（代码存储）、数据存储器和系统存储器。主要组成部分如下：

1. **程序存储器（Code Memory）**：通常是**闪存（Flash）**或其他非易失性存储介质，用于存储程序代码。程序存储器通常是只读的，除非特定操作。
2. **数据存储器（Data Memory）**：数据存储器一般使用**静态随机存储器（SRAM）**，用于存储程序执行时使用的数据（如全局变量、局部变量、堆栈等）。
3. **外设存储器映射（Memory-Mapped I/O）**：Cortex-M3通过**内存映射I/O**（MMIO）机制与外设交互，所有外设寄存器（如GPIO、定时器、ADC等）都被映射到一个特定的内存地址区间，通过读取或写入这些地址可以与外设进行通信。
4. **堆栈和堆（Stack and Heap）**：Cortex-M3通过堆栈（stack）和堆（heap）管理内存：
    - **堆栈**：用于存储局部变量、函数参数和返回地址等临时数据。

- **堆**：用于动态分配内存，通常在程序运行时由操作系统或程序员管理。

5. **内存保护单元（MPU）**：内存保护单元（MPU）是Cortex-M3的一个可选组件，用于设置内存区域的访问权限，确保程序的内存安全。

2.3.4 中断与异常（NVIC）

Cortex-M3内核提供了一个高效的中断控制系统，称为**嵌套向量中断控制器（NVIC）**，用于处理和管理硬件中断和异常。NVIC是Cortex-M3内核的一部分，支持快速响应和灵活的中断处理。

1. **NVIC功能**：
   - **中断优先级管理**：NVIC允许配置中断的优先级，支持多达256个中断源。每个中断源可以设置一个优先级，优先级越高的中断会被优先响应。
   - **中断使能和禁用**：可以通过NVIC使能或禁用特定的中断源。
   - **中断嵌套**：支持中断嵌套，允许较高优先级的中断打断低优先级的中断服务程序。
   - **中断清除和确认**：处理完中断后，NVIC会自动清除或确认该中断源的状态。
2. **中断处理流程**：
   - 当中断请求发生时，处理器检查当前执行的指令，如果没有其他更高优先级的中断，则将处理器状态保存到堆栈中，并跳转到中断处理程序。
   - 中断处理程序执行完毕后，使用**中断返回指令（IRET）**恢复先前的程序状态，返回到中断发生前的代码执行位置。
3. **中断分组**：NVIC允许根据不同的优先级设置中断的分组，支持较细粒度的优先级控制。例如，可以在不同的组之间设置优先级，确保重要的任务不会被低优先级的中断打断。

2.4 STM32微控制器结构

STM32微控制器采用高效的ARM Cortex-M系列内核，具有丰富的外设和灵活的系统架构。除了核心的处理器内核，STM32还包括多个子系统，确保其在多种应用场景中都能高效地运行。以下是STM32系统的详细结构内容。

2.4.1 STM32系统结构

STM32系统结构的核心包括处理器内核、内存、外设和其他重要模块。STM32的设计使其在提供高性能的同时也能够满足低功耗和灵活的应用需求。其主要组成部分如下：

1. **处理器内核**：
   - STM32微控制器基于**ARM Cortex-M系列内核**（如Cortex-M0/M3/M4/M7等）。不同的内核有不同的性能特点，Cortex-M0适用于低功耗应用，而Cortex-M4/M7则适合需要更强计算能力和数字信号处理的应用。
   - **指令集架构**：ARM Cortex-M系列使用精简指令集（RISC），支持Thumb-2指令集，能提供较低的代码大小和较高的执行效率。
2. **内存管理**：
   - **Flash存储器**：用于存储程序代码，通常为非易失性存储，容量范围从几十KB到几MB不等。
   - **SRAM**：静态随机存储器，用于存储运行时数据，如堆栈和全局变量。
   - **外部存储器**：某些STM32型号支持外部Flash、SRAM或其他外设存储器。
3. **外设接口**：STM32提供丰富的外设接口，包括GPIO、USART、I2C、SPI、CAN、USB等，满足各种传感器、通信设备和其他外设的连接需求。
4. **中断管理**：STM32微控制器内置**NVIC**（嵌套向量中断控制器），支持高效的中断和异常管理。

5. **低功耗管理**：STM32支持多种低功耗模式（如睡眠模式、待机模式等），适合电池供电的应用。

2.4.2 STM32总线结构

STM32微控制器采用了高效的总线结构，确保各个模块之间能够快速、高效地通信。STM32的总线结构主要包括：

1. **系统总线（AHB）**：
   - **AHB总线（Advanced High-performance Bus）**是STM32的主总线，连接处理器内核、内存和高带宽的外设（如DMA控制器、外部存储器接口等）。
   - AHB总线提供高速数据传输，支持高带宽需求的模块。
2. **外设总线（APB）**：
   - **APB总线（Advanced Peripheral Bus）**用于连接低带宽外设，如定时器、GPIO、UART、I2C等。APB总线的工作频率通常较低，适用于对带宽要求不高的外设。
   - STM32支持APB1和APB2两种总线，APB2通常用于更高速的外设，如SPI和USART，而APB1则连接较慢的外设，如低速定时器。
3. **外部总线接口（FSMC）**：
   - 在某些STM32型号中，提供外部存储器控制器（FSMC，Flexible Static Memory Controller）来连接外部Flash、SRAM等外部存储器。FSMC具有灵活的配置选项，适用于扩展存储需求的应用。
4. **系统互连**：
   - STM32的各个模块通过高效的互连方式进行通信，保证内存、外设和DMA之间的高效数据流。

2.4.3 STM32存储结构

STM32微控制器的存储结构设计考虑了处理器、外设和低功耗模式的需求，提供了灵活而高效的内存管理。STM32的存储结构包括以下几个部分：

1. **闪存（Flash）**：
   - 用于存储程序代码和部分常量数据，闪存是非易失性的，可以在断电后保留数据。
   - STM32的闪存容量从几十KB到几MB不等，某些型号支持外部闪存接口。
2. **SRAM**：
   - 静态随机存储器（SRAM）用于存储运行时数据，如局部变量、堆栈、堆内存等。SRAM是易失性的，在断电后数据丢失。
   - STM32提供的SRAM容量根据型号不同，通常为几十KB到几百KB。
3. **外部存储器**：
   - 在需要大容量存储的应用中，STM32可以通过外部存储器接口（如FSMC）连接外部存储器（如外部SRAM、Flash等），扩展存储容量。
4. **中断向量表**：
   - STM32内核使用中断向量表来存储中断服务程序的地址。该表位于闪存中，并通过特殊的指令机制进行访问。
5. **内存保护单元（MPU）**：
   - 部分STM32微控制器支持内存保护单元（MPU），可以在运行时对内存进行保护，防止非法访问或数据损坏。MPU能够划分不同的内存区域并设置访问权限。

2.4.4 STM32中断

STM32微控制器采用嵌套向量中断控制器（NVIC）来管理系统的中断和异常。NVIC提供了快速的中断响应、优先级控制和中断嵌套能力。其主要特性如下：

1. **中断源**：
    - STM32支持多达256个中断源，包括外部中断、内部外设中断和系统异常等。每个中断源都有一个唯一的标识符。
2. **中断优先级**：
    - NVIC允许为每个中断源设置优先级，并且支持中断优先级嵌套。中断优先级越高的中断会优先响应，低优先级的中断可以被高优先级的中断打断。
3. **中断使能和禁用**：
    - 可以通过NVIC使能或禁用特定的中断源。禁用中断后，相关中断请求不会触发中断服务程序。
4. **中断触发方式**：
    - STM32支持不同的中断触发方式，包括边沿触发（例如，GPIO引脚状态变化）和电平触发（例如，外部中断信号保持某一电平）。
5. **中断嵌套**：
    - STM32支持中断嵌套，允许高优先级的中断中断低优先级的中断服务程序。通过设置中断优先级，可以灵活管理中断处理顺序。
6. **异常管理**：
    - 除了外部中断，STM32还能够管理硬件异常（如硬件故障）、系统异常（如内存访问错误）和软件中断。

2.4.5 STM32时钟系统

STM32微控制器的时钟系统是整个芯片运行的核心，负责为处理器和外设提供时钟信号。时钟系统的结构包括多个时钟源和时钟分配模块，能够为不同模块提供适当的时钟频率。主要组成部分如下：

1. **时钟源**：
    - **内部高速时钟（HSI）**：用于提供处理器和系统模块的时钟源，频率通常为8MHz。
    - **外部高速时钟（HSE）**：通过外部晶振提供高精度的时钟源，频率通常为8MHz或更高。
    - **内部低速时钟（LSI）**：用于低功耗模式下的时钟源，频率通常为32kHz。
    - **外部低速时钟（LSE）**：通过外部晶振提供低精度时钟源，通常用于实时时钟（RTC）功能。
2. **时钟分配**：
    - STM32通过时钟树（Clock Tree）管理不同模块的时钟分配，确保系统、外设和低功耗模块能够获得适当的时钟信号。
    - 支持通过**PLL（Phase-Locked Loop）**对时钟进行倍频，以提高处理器和外设的时钟频率。
3. **低功耗模式**：
    - STM32支持多种低功耗模式，可以关闭某些时钟源以减少功耗。例如，进入**待机模式**时，CPU和部分外设会停止时钟信号，减少功耗。
4. **时钟切换**：
    - STM32可以动态切换时钟源（例如从HSI切换到HSE），以适应不同应用场景的需求。通过软件控制时钟源的切换，可以在性能和功耗之间做平衡。

总结

STM32微控制器的系统架构设计具有高度的灵活性和扩展性，能够满足从简单控制到复杂实时处理的各种应用需求。其丰富的外设支持、高效的总线架构、灵活的存储结构、强大的中断处理和精准的时钟系统使其成为嵌入式系统领域的广泛选择。

第5章 通用输入/输出 (GPIO)

5.1 GPIO概述

GPIO（General Purpose Input/Output，通用输入/输出）是微控制器中的一种基础外设，广泛应用于嵌入式系统中，用于实现与外部设备的通信和控制。STM32微控制器提供了灵活且功能强大的GPIO接口，可以配置为输入、输出、复用模式，并且支持中断、PWM输出等功能，适用于各种控制和信号处理任务。

GPIO端口是微控制器与外部世界互动的桥梁，常常用于读取开关状态、控制LED、与外部传感器通信、驱动电机等操作。根据功能需求，GPIO的工作模式可以灵活配置，以满足不同的应用场景。

5.2 STM32的GPIO

STM32微控制器提供了灵活的GPIO（通用输入/输出）功能，可用于连接各种外部设备，实现输入、输出、复用功能以及中断管理等多种操作。STM32的GPIO引脚可通过多种方式进行配置，支持多种电气特性和功能模式，广泛应用于控制LED、按钮、传感器等设备的接口。

5.2.1 GPIO引脚

STM32微控制器的GPIO引脚设计得非常灵活，每个引脚都可以配置成不同的功能模式，包括输入、输出、复用和模拟模式。每个GPIO端口（如GPIOA、GPIOB、GPIOC等）通常包含16个引脚，这些引脚通过配置寄存器来控制其功能和工作模式。

1. GPIO引脚的基本分类

STM32的GPIO引脚根据功能的不同可以分为以下几类：

- **普通I/O引脚**：用于数字信号的输入或输出。
- **复用I/O引脚**：这些引脚可以被配置为连接到STM32的外设（如USART、SPI、I2C等），用于进行通信等高级功能。
- **模拟I/O引脚**：这些引脚用于连接模拟信号，如用于ADC（模拟到数字转换）或DAC（数字到模拟转换）。
- **中断输入引脚**：某些GPIO引脚可以配置为外部中断源，用于响应外部事件

5.2.2 GPIO内部结构

STM32的GPIO内部结构包括多个组成部分，用于控制和管理引脚的功能、工作模式和电气特性。每个GPIO端口由多个引脚组成，通常是16个引脚（从0到15），每个引脚都有独立的配置和控制寄存器。GPIO的内部结构包括以下几个重要部分：

1. GPIO引脚配置寄存器（CRL/CRH）

每个GPIO端口（如GPIOA、GPIOB等）都包含两个配置寄存器：`CRL`和`CRH`，分别控制低位（0~~7）和高位~~~~（8~~15）引脚的工作模式。每个引脚有4个比特位用于配置：

- **模式选择**（2位）：选择输入、输出、复用或模拟模式。
- **输出类型**（1位）：选择推挽输出或开漏输出。
- **输出速度**（2位）：设置输出引脚的速度等级。
- **上拉/下拉电阻**（2位）：选择是否启用上拉或下拉电阻。

通过这些寄存器，可以配置每个引脚的工作模式、输出类型和速度。

2. GPIO输入数据寄存器（IDR）

`IDR`寄存器用于读取GPIO引脚的输入状态。当GPIO配置为输入模式时，可以通过该寄存器读取每个引脚的电平（高或低）。读取操作通常是对一个单独引脚的电平进行查询。

3. GPIO输出数据寄存器（ODR）

`ODR`寄存器用于控制GPIO引脚的输出状态。当GPIO配置为输出模式时，可以通过该寄存器设置每个引脚的输出电平。该寄存器的每一位对应一个GPIO引脚，1表示高电平，0表示低电平。

4. GPIO设置位寄存器（BSRR）

`BSRR`寄存器用于快速设置GPIO引脚的电平。与`ODR`不同，`BSRR`寄存器可以单独设置引脚为高电平或低电平，提供更高效的操作。通过设置不同的位（设置位和复位位），可以实现单独的高电平或低电平输出。

5. GPIO重置寄存器（BRR）

`BRR`寄存器用于设置GPIO引脚为低电平。它是`BSRR`寄存器的一部分，用于控制引脚的低电平输出。

6. GPIO锁定寄存器（LCKR）

`LCKR`寄存器用于锁定GPIO的配置，防止在配置过程中被更改。当需要确保GPIO配置不会被意外修改时，可以使用锁定寄存器来防止更改。

7. GPIO中断寄存器（EXTI）

STM32允许通过GPIO引脚触发外部中断，`EXTI`寄存器用于管理这些中断。每个GPIO引脚可以配置为外部中断源，并根据外部信号的变化触发中断。中断触发方式（上升沿、下降沿或双边沿）可以通过配置寄存器来选择。

5.2.3 GPIO工作模式

STM32的GPIO引脚可以配置为多种工作模式，以下是主要的工作模式及其用途：

1. 输入模式

输入模式用于读取外部信号。GPIO引脚在输入模式下可以有三种不同的配置：

- **浮空输入（Floating Input）**：输入引脚不接任何电平（高电平或低电平），即该引脚不进行上拉或下拉，保持高阻抗状态。通常用于与外部设备连接时，不需要电平固定的情况。
- **上拉输入（Pull-up Input）**：输入引脚通过内部上拉电阻连接到Vdd电源，默认情况下将引脚拉高。适用于需要始终保持引脚高电平，除非外部电路将其拉低的情况。
- **下拉输入（Pull-down Input）**：输入引脚通过内部下拉电阻连接到GND，默认情况下将引脚拉低。适用于需要始终保持引脚低电平，除非外部电路将其拉高的情况。

2. 输出模式

输出模式用于将GPIO引脚配置为输出信号，控制外部设备（如LED、继电器、电机等）。输出模式有两种主要配置：

- **推挽输出（Push-pull）**：引脚在输出逻辑高时驱动Vdd电压（高电平），在输出逻辑低时驱动GND电压（低电平）。这种模式适用于普通的数字信号输出。

- **开漏输出（Open-drain）**：引脚仅在输出逻辑低时驱动GND电压，而在输出逻辑高时处于高阻抗状态。适用于多个设备共用同一信号线的场景（例如I2C总线），引脚通过外部拉高电阻来维持高电平。

3. 复用模式

复用模式允许GPIO引脚连接到STM32的内部外设接口，如USART、SPI、I2C等。配置复用模式时，需要选择适当的外设功能来连接引脚。复用功能通过AF（Alternate Function）寄存器进行配置，允许每个引脚支持多个外设功能。

4. 模拟模式

某些GPIO引脚支持模拟模式，允许读取模拟信号（如传感器输出），并通过内置的ADC（模拟到数字转换器）将其转换为数字信号。在模拟模式下，GPIO引脚不进行数字输入输出操作，而是直接与模拟电路交互。

5.2.4 GPIO输出速度

STM32的GPIO引脚在配置为输出模式时，允许设置输出引脚的速度等级。输出速度等级控制GPIO引脚在输出高电平时的切换速率，影响信号的上升沿和下降沿的速度。速度设置有三个等级：

1. 低速（Low Speed）

低速模式下，引脚的切换速率较慢，适用于低频或低速的输出信号。这种设置通常会减小功耗，并且用于驱动小电流负载。

2. 中速（Medium Speed）

中速模式下，引脚的切换速率比低速模式快，适用于需要较快响应但又不需要极高速率的应用。适合一般的数字信号输出。

3. 高速（High Speed）

高速模式下，引脚的切换速率最快，适用于需要快速切换信号的场景。该模式通常用于高速信号传输和需要较大驱动能力的应用，如驱动LED、PWM信号输出等。

选择合适的输出速度可以根据应用的需求平衡响应速度和功耗，同时还要考虑到所驱动负载的特性。

总结

STM32的GPIO引脚具有非常灵活的配置选项，通过合理的配置寄存器，可以选择输入、输出、复用或模拟模式，控制每个引脚的工作行为。GPIO的输出速度、输入模式、复用功能等都可以根据应用需求进行调整。理解GPIO的内部结构和工作模式对开发高效稳定的嵌入式系统至关重要。

5.3 GPIO标准外设库接口函数及应用

5.3.1 GPIO标准外设库接口函数

STM32的标准外设库为GPIO提供了一系列接口函数，用于配置、操作和管理GPIO引脚。通过这些函数，用户可以方便地控制GPIO的输入输出、配置工作模式、设置中断等。标准外设库函数大大简化了直接操作寄存器的复杂度，并提供了高层次的抽象，使得开发更加便捷。

常见的GPIO标准外设库接口函数：

1. **GPIO初始化函数：**

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
```

功能： 配置GPIO引脚的模式、输出类型、速度、上拉/下拉电阻等。

参数：

- `GPIOx`：要配置的GPIO端口，例如GPIOA、GPIOB等。
- `GPIO_InitStruct`：配置参数结构体，包含了引脚的配置参数（例如工作模式、输出类型、输出速度等）。

2. **GPIO设置输出电平函数：**

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

功能： 设置指定引脚为高电平。

参数：

- `GPIOx`：要设置的GPIO端口。
- `GPIO_Pin`：要设置的引脚编号（如GPIO_Pin_0、GPIO_Pin_1等）。

示例： 设置GPIOA的第5引脚为高电平：

```
GPIO_SetBits(GPIOA, GPIO_Pin_5);
```

3. **GPIO清除输出电平函数：**

```
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

功能： 设置指定引脚为低电平。

参数：

- `GPIOx`：要设置的GPIO端口。
- `GPIO_Pin`：要设置的引脚编号。

示例： 设置GPIOA的第5引脚为低电平：

```
GPIO_ResetBits(GPIOA, GPIO_Pin_5);
```

4. **GPIO读取输入电平函数：**

```
BitAction GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

功能： 读取指定引脚的输入电平。

参数：

- `GPIOx`：要读取的GPIO端口。
- `GPIO_Pin`：要读取的引脚编号。

返回值：

- `Bit_SET`：引脚为高电平。
- `Bit_RESET`：引脚为低电平。

示例： 读取GPIOA的第5引脚的输入电平：

```
if (GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_5) == Bit_SET)
{
    // 引脚为高电平
}
else
{
    // 引脚为低电平
}
```

5. **GPIO配置中断函数：**

```
void GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
```

功能： 配置外部中断线路，以便外部事件触发中断。

参数：

- `GPIO_PortSource`：选择GPIO端口源，例如GPIO_PortSourceGPIOA、GPIO_PortSourceGPIOB等。
- `GPIO_PinSource`：选择GPIO引脚源，例如GPIO_PinSource0、GPIO_PinSource1等。

6. **GPIO设置为模拟模式：**

```
void GPIO_StructInit(GPIO_InitTypeDef* GPIO_InitStruct);
```

功能： 初始化GPIO配置结构体为默认值，便于设置模拟模式。

示例：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_StructInit(&GPIO_InitStructure);  // 设置结构体为默认值
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;  // 设置为模拟模式
GPIO_Init(GPIOA, &GPIO_InitStructure);  // 初始化GPIOA
```

7. **GPIO重映射功能：**

```
void GPIO_PinAFConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_PinSource, uint8_t GPIO_AF);
```

功能： 配置GPIO引脚复用功能。

参数：

- `GPIOx`：选择的GPIO端口（例如GPIOA、GPIOB等）。
- `GPIO_PinSource`：选择引脚编号。
- `GPIO_AF`：选择复用功能（例如USART、SPI等）。

示例： 配置GPIOA的第9引脚为USART1的TX引脚：

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1);
```

5.3.2 GPIO标准外设库应用实例

以下是一个简单的GPIO应用示例，演示如何配置GPIO引脚以控制一个LED灯。

硬件连接：

- 将LED连接到GPIOA的第5个引脚（PA5）。

步骤：

1. 配置PA5为输出模式。
2. 在主循环中不断切换PA5的电平，从而控制LED的开关。

代码示例：

```
#include "stm32f10x.h"

void GPIO_Init(void)
{
    // 启用GPIOA时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // 配置GPIOA的第5个引脚为推挽输出
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;           // 选择PA5
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;     // 配置为推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;    // 设置输出速度
    GPIO_Init(GPIOA, &GPIO_InitStructure);               // 初始化PA5
}

void Delay(uint32_t delay)
{
    while (delay--)
    {
        __NOP();  // 空操作，用于延时
    }
}

int main(void)
{
    GPIO_Init();  // 初始化GPIO

    while (1)
    {
        GPIO_SetBits(GPIOA, GPIO_Pin_5);  // 设置PA5为高电平（LED亮）
        Delay(1000000);                   // 延时
        GPIO_ResetBits(GPIOA, GPIO_Pin_5);  // 设置PA5为低电平（LED灭）
        Delay(1000000);                   // 延时
```

```
    }
}
```

在此示例中，PA5被配置为推挽输出模式。主程序中通过 `GPIO_SetBits` 和 `GPIO_ResetBits` 函数切换PA5的电平，控制LED的亮灭。

5.3.3 基于标准外设库开发的一般流程

基于STM32标准外设库开发GPIO应用程序通常遵循以下步骤：

1. 初始化硬件资源

在应用程序开始之前，首先需要初始化硬件资源（例如GPIO、时钟等）。这通常在 `main()` 函数的开头进行。

- 启用外设时钟：使用 `RCC_APB2PeriphClockCmd` 或 `RCC_AHB1PeriphClockCmd` 函数启用GPIO等外设的时钟。
- 配置GPIO引脚：使用 `GPIO_Init` 函数配置GPIO引脚的工作模式、输出类型、上拉/下拉电阻等。

2. 配置GPIO引脚

根据应用的需求配置GPIO引脚。常见配置包括设置为输入、输出、模拟模式或复用功能。可以使用 `GPIO_InitTypeDef` 结构体和 `GPIO_Init` 函数进行配置。

3. 操作GPIO引脚

根据需要的功能，使用库函数控制GPIO引脚。例如，使用 `GPIO_SetBits` 和 `GPIO_ResetBits` 设置输出引脚的高低电平，使用 `GPIO_ReadInputDataBit` 读取输入引脚的电平。

4. 中断配置（如果需要）

如果应用需要GPIO中断功能，配置外部中断线路（EXTI），并在中断服务程序（ISR）中处理相关事件。使用 `GPIO_EXTILineConfig` 和 `EXTI_Init` 函数进行配置。

5. 调试与优化

在应用程序完成后，通过调试工具（如ST-Link或J-Link）进行调试，查看GPIO操作是否符合预期。如果需要，进一步优化代码，减少延时、提高效率等。

总结

通过使用STM32标准外设库，开发者可以更加便捷地配置和控制GPIO引脚。标准库提供了丰富的接口函数，涵盖了GPIO的输入输出、复用功能、中断配置等常见操作。应用实例展示了如何使用这些函数控制LED，同时基于标准外设库的开发流程也为嵌入式系统开发提供了清晰的步骤和方法。

第6章 中断

6.1 中断的相关概念

中断是嵌入式系统中非常重要的概念，它用于应对外部事件的及时响应。中断的使用可以使系统在处理某些任务时不会一直停留在一个流程中，而是能够灵活响应外部的变化，从而提高系统的响应性和效率。

6.1.1 什么是中断

中断（Interrupt）是一种在微处理器中用于暂停当前程序执行并转而执行其他任务的机制。中断通常由硬件或软件触发，当某个特定事件发生时，处理器会暂停正在执行的程序，转而处理该事件对应的中断服务程序（ISR）。中断是一种异步机制，即它可以在程序执行的任何时刻发生。

中断的主要特点：

- **中断源：** 中断源可以是外部设备、内部事件或者定时器等。
- **中断响应：** 当中断发生时，处理器会响应中断请求，并跳转到对应的中断服务程序（ISR）去处理。
- **中断优先级：** 不同的中断可以设置不同的优先级，优先级高的中断会先被处理。

中断分为两类：

1. **外部中断：** 由外部设备或事件触发，例如外部按键按下、外部信号变化等。
2. **内部中断：** 由内部事件触发，例如定时器溢出、看门狗超时、软件中断等。

6.1.2 为什么使用中断

使用中断有以下几个主要原因：

1. **提高响应速度：** 中断允许处理器实时响应外部事件，减少了传统轮询方式的等待时间。例如，当外部设备如传感器产生变化时，使用中断可以立即响应，而不需要程序每隔一段时间轮询设备状态。
2. **节省CPU资源：** 在没有中断的情况下，CPU需要持续轮询外设以检查是否发生了事件，这会占用大量的处理时间。而使用中断时，CPU可以在空闲时进行其他任务，只有在中断发生时才中断当前工作，节省了处理器资源。
3. **提高系统实时性：** 中断机制让系统能够及时响应关键事件，这对于实时性要求较高的应用至关重要。例如，在控制系统中，响应外部传感器数据或时间中断可以影响系统的稳定性和性能。
4. **简化系统设计：** 中断机制使得程序设计更加简洁。在没有中断的情况下，程序设计可能需要复杂的轮询机制来检查事件，而中断使得这些机制变得更直观和高效。

6.1.3 中断处理流程

中断处理流程涉及从中断事件的发生到处理完成的整个过程。STM32等嵌入式系统的中断处理流程通常包括以下几个步骤：

1. **中断请求（IRQ）生成：**中断的发生通常由外部事件（如硬件设备状态变化、定时器溢出等）或内部事件（如软件触发的中断）触发。处理器持续监控这些事件，并且在发生中断请求时生成一个中断信号。
2. **中断请求的优先级判断：**如果有多个中断请求发生，系统根据中断的优先级来决定先后处理顺序。STM32中通常有硬件优先级管理机制，用户可以在初始化时设置每个中断源的优先级。
3. **中断挂起与响应：**一旦一个中断被触发，处理器会停止当前执行的程序，保存当前程序的状态（如PC寄存器、SP寄存器等），并开始执行中断服务程序（ISR）。此时，处理器进入中断处理状态，执行与该中断相关的代码。
4. **中断服务程序（ISR）执行：**中断服务程序（ISR）是针对特定中断事件的处理代码。ISR的主要任务是处理触发中断的事件，并尽量快速地执行完毕。处理完成后，ISR应该尽量简洁，不要包含阻塞性操作（如延时等），以避免影响系统的实时性。
5. **中断清除与恢复：**中断服务程序执行完成后，处理器需要清除中断标志位，以便后续中断能够被正确响应。然后，处理器会恢复到中断前的程序状态，继续执行原先的任务。
6. **中断嵌套（可选）：**在某些情况下，高优先级的中断可以打断低优先级的中断服务程序，这种机制被称为中断嵌套。在中断嵌套的情况下，中断服务程序需要更小心地保存和恢复上下文，确保每个中断的处理能够正确执行。

总结

中断是嵌入式系统中用于响应外部或内部事件的一种机制。它通过在事件发生时暂停当前程序并执行中断服务程序来提供即时响应。使用中断可以提高系统的实时性、响应速度、节省处理器资源，并简化系统设计。理解中断的概念、使用中断的原因以及中断处理流程，对于编写高效的嵌入式应用程序至关重要。

6.2 STM32中断和异常

在STM32微控制器中，中断和异常机制是处理外部和内部事件的关键工具。STM32使用中断和异常来中断程序的执行，并让系统响应各种硬件或软件的事件。了解STM32的中断和异常机制对于高效开发嵌入式系统非常重要。

6.2.1 STM32中断和异常向量表

在STM32中，所有中断和异常的入口地址都存储在一个叫做“中断向量表”（Interrupt Vector Table, IVT）的位置。中断向量表是一个包含各个中断和异常入口地址的数组，每个入口点对应一个特定的中断源或异常。

中断向量表的结构

中断向量表通常位于系统内存的起始位置，地址为 `0x00000000`（起始地址）。在STM32中，向量表通常存储在Flash或RAM中。中断向量表的每一项是一个指向中断服务程序（ISR）的地址。

中断向量表包含两类内容：

1. **复位向量（Reset Vector）：** 当微控制器上电或复位时，系统会从复位向量指定的地址开始执行程序。复位向量通常是程序的起始位置。
2. **中断向量：** 每个中断源都对应一个向量，向量表中存储每个中断的处理函数地址。中断向量是中断发生时，处理器跳转的地址。

中断向量表的组织

STM32的中断向量表按如下顺序组织：

- **复位向量：** 系统复位后处理的第一个函数地址。
- **NMI向量（Non-Maskable Interrupt）：** 处理不可屏蔽中断的函数地址。
- **硬件中断向量：** 各种硬件中断源的入口地址（如定时器、外部引脚、通信接口等）。
- **系统异常向量：** 用于处理各种系统异常（如硬错误、存储访问错误等）。

向量表的大小和内容根据具体的STM32型号和应用而有所不同，但一般来说，向量表的前几个地址是固定的，用于存储复位地址、中断源的地址以及异常服务程序。

向量表的定义

STM32使用 `startup.s` 文件（汇编语言）定义中断向量表。在此文件中，你可以看到复位向量和各个中断的入口地址。例如：

```
.section .isr_vector, "a", %progbits
.global _vectors
_vectors:
    .word _estack              /* 初始堆栈地址 */
    .word Reset_Handler        /* 复位向量 */
    .word NMI_Handler          /* NMI中断 */
    .word HardFault_Handler    /* 硬错误异常 */
    /* 其他中断向量 */
```

6.2.2 STM32中断优先级

在STM32中，中断优先级用于确定多个中断同时发生时，哪个中断应该先被处理。STM32的中断优先级机制具有一定的复杂性，允许用户配置中断的优先级，使得高优先级的中断能够打断低优先级的中断。

中断优先级的设定

STM32使用一个分为两部分的优先级系统：

1. **抢占优先级（Pre-emption Priority）：** 用于决定中断是否能打断当前的中断服务程序。抢占优先级越高的中断能越早抢占低优先级的中断。
2. **子优先级（Subpriority）：** 用于在相同抢占优先级的中断之间进一步排序。在多个中断具有相同抢占优先级时，子优先级决定哪个中断先执行。

优先级的设定是通过配置中断控制寄存器（NVIC）中的`IP`（Interrupt Priority）字段来完成的。STM32的优先级字段通常是4位的，但实际的优先级位数根据具体的型号和中断控制器的配置有所不同。

STM32中断优先级设置

STM32允许通过以下方式设置中断的优先级：

1. **中断优先级分组：** STM32支持多个中断优先级分组，分组数决定了抢占优先级和子优先级的位数。可以通过`NVIC_PriorityGroupConfig`函数设置中断优先级分组。
2. **设置具体中断的优先级：** 通过`NVIC_SetPriority`函数设置某个中断的优先级。

优先级设置示例：

```
NVIC_SetPriority(USART1_IRQn, NVIC_EncodePriority(NVIC_PriorityGroup_2, 1, 0));
```

这行代码将USART1中断的抢占优先级设置为1，子优先级设置为0，优先级分组为`NVIC_PriorityGroup_2`，这意味着抢占优先级占用2位，子优先级占用2位。

中断优先级分组

中断优先级分组决定了优先级字段中的抢占优先级和子优先级的位数。STM32支持如下几种分组：

- **NVIC_PriorityGroup_0:** 只有抢占优先级，没有子优先级。
- **NVIC_PriorityGroup_1:** 1位抢占优先级，3位子优先级。
- **NVIC_PriorityGroup_2:** 2位抢占优先级，2位子优先级。
- **NVIC_PriorityGroup_3:** 3位抢占优先级，1位子优先级。
- **NVIC_PriorityGroup_4:** 4位抢占优先级，0位子优先级。

通过`NVIC_PriorityGroupConfig()`来设置优先级分组。

6.2.3 STM32中断服务程序（ISR）

中断服务程序（ISR，Interrupt Service Routine）是当中断发生时处理器执行的函数。它是响应特定中断源的程序，通常用于处理中断事件，例如读取外设数据、清除中断标志等。

中断服务程序的工作流程

1. **中断发生：** 当中断条件满足时（例如定时器溢出、外部输入等），中断请求被发出，处理器暂停当前任务，进入中断服务程序。
2. **保存上下文：** 处理器保存当前任务的上下文（程序计数器PC、堆栈指针SP等），以便在中断结束后恢复。

3. **执行ISR：** 处理器跳转到ISR的入口点，执行ISR代码。在ISR中，通常要进行必要的中断处理，如清除中断标志、读取数据、驱动外设等。
4. **恢复上下文：** ISR执行完毕后，恢复上下文，恢复原先的任务并返回主程序。

编写中断服务程序

中断服务程序必须尽量简洁和高效，以避免占用过多的CPU时间。通常，ISR只执行最低限度的处理工作（例如清除中断标志、设置标志位等），然后尽量快地退出，避免阻塞其他中断的处理。

STM32的中断服务程序格式如下：

```
void USART1_IRQHandler(void)
{
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) // 检查是否是接收中断
    {
        uint16_t received_data = USART_ReceiveData(USART1); // 读取接收到的数据
        // 处理接收到的数据
        USART_ClearITPendingBit(USART1, USART_IT_RXNE); // 清除中断标志
    }
}
```

在ISR中，首先要判断中断标志位，确定是否触发了对应的中断，然后进行相应的处理，最后通过`USART_ClearITPendingBit`等函数清除中断标志，防止中断被重复触发。

总结

STM32的中断和异常机制允许系统在事件发生时快速响应，并通过优先级控制和中断服务程序来处理各种任务。中断向量表是中断处理的核心，STM32提供了灵活的中断优先级控制以及高效的中断服务程序设计方式。理解中断向量表、优先级管理和ISR的编写规范对于高效使用STM32中断系统至关重要。

6.3 STM32外部中断（EXTI）

STM32的外部中断（EXTI，External Interrupt）功能允许微控制器响应外部信号的变化（如按钮按下、外部传感器的信号等）。通过外部中断，STM32能够及时响应硬件事件，而无需通过轮询方式来检测设备状态，这大大提高了系统的响应速度和效率。

6.3.1 EXTI概述

EXTI是STM32的一个外设模块，用于处理外部引脚产生的中断或事件。每个EXTI通道（通常是GPIO引脚）都可以配置为响应上升沿、下降沿或双边沿（即状态变化）。EXTI模块可以与多个外部引脚（如GPIOA、GPIOB等）相连接，并通过中断机制响应这些引脚上的电平变化。

STM32的EXTI模块可以支持的功能包括：

1. **外部中断**：响应外部引脚状态的变化。
2. **事件触发**：响应外部引脚的电平变化（高电平、低电平）或边沿触发（上升沿、下降沿）。

STM32支持的外部中断引脚包括（具体的引脚依型号而异）：

- **GPIOA**、**GPIOB**、**GPIOC**等。
- 每个GPIO引脚（如PA0、PA1等）都可以通过EXTI模块配置为外部中断源。

6.3.2 EXTI的工作原理

EXTI的工作过程通常包括以下几个步骤：

1. **配置GPIO引脚**：首先，需要将某个GPIO引脚配置为输入模式，并启用外部中断功能。这通常通过配置GPIO引脚的寄存器来完成。
2. **配置EXTI模块**：通过EXTI寄存器配置外部中断的触发条件（例如，选择上升沿、下降沿或双边沿触发）。
3. **使能EXTI中断**：启用EXTI中断源，并配置中断优先级。
4. **中断服务**：当EXTI引脚状态变化时，EXTI模块触发中断，控制器跳转到中断服务程序（ISR）来处理事件。
5. **清除中断标志**：在ISR中，需要手动清除EXTI中断标志，防止中断重复触发。

6.3.3 EXTI中断配置

EXTI模块的配置主要包括以下几个方面：

- **触发条件**：选择中断的触发条件（上升沿、下降沿或双边沿）。
- **中断使能**：通过`EXTI_IMR`寄存器使能外部中断。
- **中断优先级**：通过NVIC配置中断优先级。
- **中断标志清除**：在ISR中通过清除`EXTI_PR`寄存器中的相应标志位来防止中断重复触发。

EXTI中断配置代码示例

```
c


复制代码
void EXTI0_IRQHandler(void)
{
    // 检查EXTI0中断是否发生
    if (EXTI_GetITStatus(EXTI_Line0) != RESET)
    {
        // 处理中断，例如读取外部传感器值
        // 清除EXTI0中断标志
        EXTI_ClearITPendingBit(EXTI_Line0);
    }
}
```


在上述示例中，当EXTI0引脚的状态变化触发中断时，程序进入`EXTI0_IRQHandler`函数进行处理中，并通过`EXTI_ClearITPendingBit`清除中断标志，防止中断被多次触发。

6.4 EXTI标准外设库接口函数及应用

在STM32的开发中，EXTI（外部中断）功能允许系统快速响应外部事件，如按钮按下、外部传感器的信号等。为了简化EXTI的配置与应用，STM32提供了标准外设库接口函数来帮助开发者快速实现外部中断功能。接下来将介绍标准外设库接口函数、编程步骤以及一些应用实例。

6.4.1 EXTI标准外设库接口函数

STM32的标准外设库提供了多种API函数，用于配置和管理EXTI模块。这些函数可以帮助开发者轻松实现外部中断的功能。

常用的EXTI标准外设库接口函数

1. **EXTI_Init()**
   - **功能**：用于初始化EXTI线的配置（选择中断引脚、设置触发条件等）。
   - 原型：

```
void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct);
```

2. **EXTI_ClearITPendingBit()**
   - **功能**：清除指定EXTI线的中断挂起标志。中断挂起标志是在中断发生后设置的，必须清除它才能防止中断重复触发。
   - 原型：

```
void EXTI_ClearITPendingBit(uint32_t EXTI_Line);
```

3. **EXTI_GenerateSWInterrupt()**
   - **功能**：生成一个软件触发的中断。该函数用于通过软件触发中断。
   - 原型：

```
void EXTI_GenerateSWInterrupt(uint32_t EXTI_Line);
```

4. **EXTI_ITConfig()**
   - **功能**：使能或禁用EXTI中断。可以通过该函数控制EXTI中断源的使能或禁用。
   - 原型：

```
void EXTI_ITConfig(uint32_t EXTI_Line, FunctionalState NewState);
```

5. **EXTI_LineConfig()**
   - **功能**：配置EXTI线与对应的GPIO引脚之间的映射。使用该函数可以将GPIO引脚与EXTI中断线进行连接。
   - 原型：

```
void EXTI_LineConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

EXTI_InitTypeDef 结构体

`EXTI_InitTypeDef`结构体是用于配置EXTI的结构体，包含了各种设置EXTI行为的字段。常见的字段包括：

- `EXTI_Line`：配置的EXTI线（如`EXTI_Line0`、`EXTI_Line1`等）。
- `EXTI_Mode`：配置中断模式（如`EXTI_Mode_Interrupt`或`EXTI_Mode_Event`）。
- `EXTI_Trigger`：配置中断触发方式（如`EXTI_Trigger_Rising`、`EXTI_Trigger_Falling`、`EXTI_Trigger_Rising_Falling`）。
- `EXTI_LineCmd`：配置EXTI线是否使能。

例如，配置EXTI0中断触发上升沿：

```
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line = EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

6.4.2 EXTI标准外设库中断应用编程步骤

在STM32中使用外部中断时，需要按一定的步骤进行配置。以下是使用EXTI的标准外设库实现外部中断的基本步骤：

步骤1：配置GPIO引脚

首先，必须将需要作为中断源的GPIO引脚配置为输入模式，并使能相应的时钟。通常，通过上拉或下拉配置GPIO引脚，以确保其具有有效的电平。

```
GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE); // 使能GPIOA时钟
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;             // 配置PA0为中断输入
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;         // 配置为上拉输入
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

步骤2：配置EXTI外部中断线

EXTI模块将GPIO引脚与中断线连接，需要配置GPIO与EXTI线之间的映射。此操作通过`GPIO_EXTILineConfig()`函数完成。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); // 使能AFIO时钟
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0); // 配置PA0引脚为EXTI0中断源
```

步骤3：配置EXTI触发条件

接下来，通过配置`EXTI_InitTypeDef`结构体来设置EXTI中断触发的条件，如选择上升沿、下降沿或双边沿触发。

```
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line = EXTI_Line0;               // 配置EXTI0
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;       // 配置为中断模式
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;    // 配置为上升沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE;                 // 使能EXTI0
EXTI_Init(&EXTI_InitStructure);
```

步骤4：配置中断优先级和NVIC

最后，配置中断优先级并使能NVIC中断。确保在中断发生时，处理器能够跳转到正确的中断处理函数。

```
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;          // 配置EXTI0中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;  // 设置抢占优先级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;        // 设置子优先级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;           // 使能NVIC中断
NVIC_Init(&NVIC_InitStructure);
```

步骤5：编写中断服务程序

当中断条件满足时，EXTI0中断会被触发并跳转到中断处理函数（`EXTI0_IRQHandler`）。在中断服务程序中，必须清除中断标志，以防止中断被重复触发。

```
void EXTI0_IRQHandler(void)
{
    if (EXTI_GetITStatus(EXTI_Line0) != RESET)  // 检查EXTI0中断是否触发
    {
        // 处理外部中断，例如读取传感器数据
        EXTI_ClearITPendingBit(EXTI_Line0);    // 清除EXTI0中断标志
    }
}
```


6.4.3 EXTI标准外设库应用实例

以下是一个简单的外部中断应用实例，通过配置PA0引脚来响应按钮按下事件，当按钮按下时触发中断，LED灯的状态发生变化。

应用实例：按键中断控制LED

硬件配置：

- PA0引脚连接到一个按钮。
- PD12引脚连接到一个LED。

代码实现：

```
#include "stm32f10x.h"

// 按钮和LED的GPIO配置
void GPIO_Config(void)
```

```
{
    GPIO_InitTypeDef GPIO_InitStructure;

    // 使能GPIOA和GPIOD时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOD, ENABLE);

    // 配置PA0为输入模式（按钮）
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;  // 配置为上拉输入
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 配置PD12为推挽输出模式（LED）
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;  // 配置为推挽输出
    GPIO_Init(GPIOB, &GPIO_InitStructure);
}

// 外部中断配置
void EXTI_Config(void)
{
    EXTI_InitTypeDef EXTI_InitStructure;

    // 配置EXTI Line0连接到PA0
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

    // 配置EXTI中断
    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;  // 配置为上升沿触发
```

EXTI_InitStructure.EXTI_LineCmd = ENABLE;EXTI_Init(&EXTI_InitStructure);

```
// 配置NVIC中断
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
}
// 中断服务程序
void EXTI0_IRQHandler(void)
{
if (EXTI_GetITStatus(EXTI_Line0) != RESET)
{// 按钮按下时，切换LED的状态
GPIOB->ODR ^= GPIO_Pin_12;  // 切换PD12引脚（LED状态）
    // 清除EXTI中断标志
    EXTI_ClearITPendingBit(EXTI_Line0);
}
}

int main(void){// 配置GPIO和外部中断GPIO_Config();
```

```
    EXTI_Config();
    while (1)
    {
        // 主循环为空，外部中断处理LED状态
    }
}
```

在这个实例中，我们使用PA0引脚连接到按钮，当按钮被按下时，PA0的状态会发生变化，从而触发EXTI中断。中断处理程序`EXTI0_IRQHandler`中，LED状态被切换（打开或关闭）。

总结

通过标准外设库接口函数，可以方便地配置STM32的EXTI模块来实现外部中断功能。配置步骤包括GPIO引脚设置、EXTI线配置、触发条件设置、中断优先级配置和编写中断服务程序。通过外部中断，STM32能够高效地响应外部事件，提高系统的实时性和响应速度。

第7章 串口通信

在嵌入式系统中，串口通信是最常见的数据传输方式之一，广泛应用于微控制器与其他设备之间的通信。串口通信的优势包括实现简单、成本低廉和可靠性高。本章将介绍串口通信的基本概念、异步串行通信协议及其接口配置。

7.1 通信概述

串口通信（Serial Communication）是一种常用的通信方式，它通过单根数据线按位传输数据，相较于并行通信，占用的引脚较少，通信距离较远，应用广泛。串口通信一般可以分为两种类型：

1. **异步串行通信**：不需要时钟信号，数据的传输依赖于预设的波特率。
2. **同步串行通信**：需要时钟信号，数据传输通过时钟信号同步。

在嵌入式系统中，异步串行通信被广泛使用，它的特点是简单、容易实现，并且适用于很多低速数据传输场景。

7.2 异步串行通信

异步串行通信（Asynchronous Serial Communication）是一种不需要同步时钟的通信方式。数据通过一系列的位（bit）在时间上顺序传输，通常使用起始位、数据位、校验位和停止位进行编码。接收端按照预定的波特率对数据流进行解码，从而还原出原始数据。

7.2.1 异步串行通信协议

异步串行通信协议是一套标准的约定，用于确保数据传输的正确性。常见的异步串行通信协议包括RS-232、RS-485和TTL串口通信等。协议规定了数据的格式、波特率、数据位、校验位和停止位等参数。

1. **数据格式**：
    - **起始位**：在数据传输的开始，起始位用来告诉接收端数据的开始位置。通常为1位，逻辑低电平。
    - **数据位**：数据本身是通过多个数据位传输的。常见的传输位数为5、6、7或8位。最常见的是8位数据传输。
    - **校验位**（可选）：校验位用于检测传输过程中的错误。常见的校验方式有无校验、奇校验和偶校验。
    - **停止位**：停止位用来表示数据的结束，通常为1位、1.5位或2位。

常见的串口帧格式为：

```
| 起始位 | 数据位 | 校验位 | 停止位 |
|   1    |   8    |   1    |   1    |
```

在此格式中，起始位为1位，数据位为8位，校验位为1位（可选），停止位为1位。

2. **波特率**：波特率是数据传输的速率，通常以“波特（bps）”表示。例如，9600波特率表示每秒钟传输9600位数据。发送端和接收端必须设置相同的波特率，以保证数据的正确接收。
3. **传输模式**：
   - **全双工**：可以同时进行数据的双向传输，常见于RS-232和TTL接口。
   - **半双工**：数据传输是单向的，但可以在两个方向之间切换，常见于RS-485。

7.2.2 异步串行通信接口

在STM32等微控制器中，异步串行通信通常使用**USART**（通用同步异步收发传输器）或**UART**（通用异步收发传输器）模块来实现。USART和UART模块在硬件上为串口通信提供了必要的功能，如起始位、数据位、停止位和波特率控制等。

1. USART和UART的区别：

- **USART**：支持同步和异步通信，因此除了异步模式外，还可以进行同步模式下的通信。
- **UART**：通常指的是异步串行通信接口，它仅支持异步模式。

在STM32微控制器中，USART模块支持异步串行通信，并通过标准外设库提供了一些接口函数，方便开发者配置和使用。

1. USART配置步骤

以STM32的USART1为例，常见的配置步骤如下：

1. **使能USART时钟**：通过使能USART外设的时钟，可以启动串口通信模块。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
```

2. **配置GPIO引脚**：配置USART的TX（发送端）和RX（接收端）引脚为复用模式。

```
GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);  // 使能GPIOA时钟
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;              // USART1 TX引脚
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;         // 复用推挽输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;       // 引脚速度设置
GPIO_Init(GPIOA, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;             // USART1 RX引脚
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;   // 浮空输入
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

3. **配置USART参数**：设置USART的工作模式，如波特率、数据位、停止位和校验位等。

```
USART_InitTypeDef USART_InitStructure;
USART_InitStructure.USART_BaudRate = 9600;            // 设置波特率
USART_InitStructure.USART_WordLength = USART_WordLength_8b; // 8位数据长度
USART_InitStructure.USART_StopBits = USART_StopBits_1; // 1位停止位
USART_InitStructure.USART_Parity = USART_Parity_No;    // 无校验位
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None; // 无硬件流控制
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx; // 配置为收发模式
USART_Init(USART1, &USART_InitStructure);
```

4. **使能USART**：启用USART模块，开始进行数据传输。

```
USART_Cmd(USART1, ENABLE);  // 启用USART1
```

2. USART数据发送和接收

1. **发送数据**：使用`USART_SendData()`函数发送数据。

```
USART_SendData(USART1, 'A');  // 发送字符'A'
while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);  // 等待发送缓冲区空
```

2. **接收数据**：使用`USART_ReceiveData()`函数接收数据。

```
if (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET)  // 检查是否接收到数据
{
    char receivedData = USART_ReceiveData(USART1);  // 读取接收到的数据
}
```

总结

异步串行通信是一种非常常见的通信方式，它通过标准的协议格式传输数据，适用于微控制器与外部设备之间的通信。STM32通过USART模块支持异步串行通信，开发者可以通过配置GPIO引脚和USART外设来实现数据的收发。理解异步串行通信协议以及如何配置USART接口，对于设计高效的嵌入式系统至关重要。

7.3 STM32的USART模块

在STM32微控制器中，USART（通用同步异步收发传输器）模块用于实现串行通信，支持异步（UART）和同步模式。USART模块广泛应用于数据传输、调试、通信协议的实现等场景。理解USART模块的内部结构、接口及编程模式对有效利用STM32进行串口通信至关重要。

7.3.1 USART内部结构

USART模块的内部结构设计用于支持异步和同步串行通信。STM32的USART模块一般由以下几个关键组成部分构成：

1. **数据寄存器（DR，Data Register）**：
    - 用于存储发送和接收的数据。USART模块通过该寄存器与外部设备进行数据交换。

- 发送数据时，将数据写入 `DR` 寄存器，USART将其移出并通过TX线发送。
- 接收数据时，数据从RX线接收并存储到 `DR` 寄存器中，供CPU读取。

2. **控制寄存器（CR1、CR2、CR3）**：
   - **CR1**：控制USART的工作模式，配置数据位、停止位、波特率、校验等。
   - **CR2**：配置停止位数、LIN协议等。
   - **CR3**：控制硬件流控制、DMA模式等。
3. **状态寄存器（SR，Status Register）**：
   - 用于标志USART模块的当前状态，如是否接收或发送数据、是否出现错误等。常用标志位包括：
     - `TXE`：发送缓冲区空标志。
     - `RXNE`：接收缓冲区非空标志。
     - `TC`：传输完成标志。
     - `ORE`：溢出错误标志。
4. **波特率生成器（BRR，Baud Rate Register）**：
   - 该寄存器用于设置USART的波特率。波特率由外部时钟频率（如APB时钟）与BRR寄存器的值共同确定。
5. **控制与数据流管理**：
   - **USART的发送与接收寄存器**：实现数据的串行传输。
   - **DMA传输支持**：USART模块可与DMA（直接存储器访问）协同工作，自动处理数据传输，提高数据处理效率。
6. **中断控制**：
   - USART模块内建中断支持，当数据准备好时（如接收到数据或发送完成），会触发中断，程序可以通过中断服务程序（ISR）处理数据。

7.3.2 USART接口

STM32的USART模块提供了多种接口功能，用于连接外部设备并进行串行通信。常见的USART接口包括：

1. **TX（传输）和RX（接收）引脚**：
   - **TX引脚**：用于发送数据。
   - **RX引脚**：用于接收数据。
2. **硬件流控制**：
   - STM32的USART模块支持硬件流控制，如RTS（请求发送）和CTS（清除发送），这些信号用于在串行通信中提供流量控制。
     - **RTS（Request to Send）**：在串行通信中，主设备可以通过RTS信号告知从设备准备发送数据。
     - **CTS（Clear to Send）**：从设备通过CTS信号告诉主设备可以开始发送数据。
3. **同步与异步模式**：
   - **异步模式**（UART）：没有时钟线，数据通过波特率同步发送，适用于一般的低速通信。
   - **同步模式**：需要时钟信号，数据在时钟信号的控制下同步发送，适用于高速通信。
4. **多路复用（GPIO复用）**：
   - USART的TX/RX引脚通常为复用引脚，可以通过修改GPIO的复用功能来选择这些引脚。比如，STM32的PA9和PA10引脚可以作为USART1的TX和RX引脚。

例如，配置PA9和PA10为USART1的TX和RX：

```
GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;  // 设置为复用推挽输出模式
GPIO_Init(GPIOA, &GPIO_InitStructure);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;  // 设置为浮空输入模式
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

5. **中断引脚**：
    - STM32的USART模块支持多种中断模式，如接收中断（RXNE），发送中断（TXE）等。当USART接收到数据或发送完成时，会触发中断。

7.3.3 USART编程模式

USART模块的编程模式包括配置USART外设、设置波特率、数据位、停止位、校验位、启用/禁用接收和发送中断等。开发者通常通过设置USART的控制寄存器（CR1、CR2、CR3）来控制USART模块的行为。

1. 配置USART波特率：

波特率是串行通信的基础，决定了数据传输的速率。在STM32中，波特率通常通过BRR寄存器配置。波特率由时钟频率（如APB时钟）和BRR寄存器的值共同确定。

```
USART_BRR = (APB_CLOCK / USART_BAUDRATE);
```

2. 配置USART传输模式（同步/异步）：

- **异步模式**：配置`USART_CR1`寄存器中的`USART_MODE`为发送/接收模式。
- **同步模式**：配置同步模式时需要额外的时钟线和时钟极性设置。

3. 启用USART模块：

使用`USART_Cmd`函数启用USART模块，使能数据的发送与接收。

```
USART_Cmd(USART1, ENABLE);
```

4. 发送数据：

发送数据时，可以将数据写入数据寄存器（`DR`），并通过TX引脚发送数据。发送完成后，TXE标志会被置为1，表示发送缓冲区为空，准备发送下一个数据。

```
USART_SendData(USART1, data);
while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
```

5. 接收数据：

接收数据时，接收到的数据会存储在`DR`寄存器中，RXNE标志被置为1，表示数据已准备好被读取。

```
if (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET) {
    receivedData = USART_ReceiveData(USART1);
}
```

6. 启用中断模式：

STM32的USART模块可以配置为中断模式，以便在数据接收或发送完成时触发中断。启用中断后，需要编写中断服务程序来处理数据。

启用接收中断：

```
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); // 启用接收中断
```

USART中断服务程序（接收数据）：

```
void USART1_IRQHandler(void)
{
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        uint8_t receivedByte = USART_ReceiveData(USART1);
        // 处理接收到的数据
        USART_ClearITPendingBit(USART1, USART_IT_RXNE); // 清除中断标志
    }
}
```

7. 错误检测：

USART模块提供了错误检测功能，可以检测到如溢出错误（ORE）、帧错误（FE）等。发生错误时，需要清除错误标志并做相应处理。

```
if (USART_GetFlagStatus(USART1, USART_FLAG_ORE) != RESET) {
    USART_ClearFlag(USART1, USART_FLAG_ORE); // 清除溢出错误标志
}
```

8. DMA模式：

USART还支持与DMA一起工作，可以在不使用中断的情况下进行数据传输。这对于需要高效传输大量数据的应用非常有用。

启用USART的DMA支持：

```
USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE);  // 启用DMA发送
USART_DMACmd(USART1, USART_DMAReq_Rx, ENABLE);  // 启用DMA接收
```

总结

STM32的USART模块是嵌入式系统中常用的串行通信

7.4 USART标准外设库接口函数及应用

STM32的USART模块提供了标准外设库接口函数，简化了USART的配置与操作，帮助开发者高效地进行串口通信应用的开发。标准外设库提供的接口函数可以帮助配置USART的波特率、数据格式、发送与接收功能、流控制等。下面我们将详细介绍USART标准外设库接口函数及其应用实例。

7.4.1 USART标准外设库接口函数

USART的标准外设库提供了多种接口函数用于配置和操作USART模块。以下是一些常用的USART库函数。

1. USART初始化与配置

- `USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct)` 配置USART的波特率、数据位、停止位、校验位、传输模式等。

```
USART_InitTypeDef USART_InitStructure;
USART_InitStructure.USART_BaudRate = 9600;            // 设置波特率为9600
USART_InitStructure.USART_WordLength = USART_WordLength_8b; // 8位数据
USART_InitStructure.USART_StopBits = USART_StopBits_1;    // 1位停止位
USART_InitStructure.USART_Parity = USART_Parity_No;        // 无校验位
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx; // 启用发送与接收功能
USART_Init(USART1, &USART_InitStructure);
```

- `USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState)` 启用或禁用USART外设。

```
USART_Cmd(USART1, ENABLE); // 启用USART1
```

2. USART发送与接收数据4

- `USART_SendData(USART_TypeDef* USARTx, uint16_t Data)` 发送一个字节的数据。

```
USART_SendData(USART1, 'A'); // 发送字符'A'
```

- `USART_ReceiveData(USART_TypeDef* USARTx)` 接收一个字节的数据。

```
uint8_t receivedData = USART_ReceiveData(USART1); // 接收数据
```

3. USART标志和中断

- `USART_GetFlagStatus(USART_TypeDef* USARTx, USART_FLAG Flag)` 获取USART模块的标志位状态，如接收缓冲区是否为空（RXNE）等。

```
if (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET) {
    uint8_t receivedByte = USART_ReceiveData(USART1); // 接收数据
}
```

- `USART_ITConfig(USART_TypeDef* USARTx, USART_IT USART_IT, FunctionalState NewState)` 配置USART中断。

```
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); // 启用接收中断
```

- `USART_ClearITPendingBit(USART_TypeDef* USARTx, USART_IT USART_IT)` 清除USART中断待处理标志。

```
USART_ClearITPendingBit(USART1, USART_IT_RXNE); // 清除接收中断标志
```

4. 波特率设置

- `USART_SetBaudRate(USART_TypeDef* USARTx, uint32_t BaudRate)`

  设置USART波特率。

```
USART_SetBaudRate(USART1, 9600); // 设置波特率为9600
```

5. 流控制

- `USART_HardwareFlowControlCmd(USART_TypeDef* USARTx, FunctionalState NewState)`

  启用或禁用USART的硬件流控制（RTS/CTS）。

```
USART_HardwareFlowControlCmd(USART1, ENABLE); // 启用硬件流控制
```

7.4.2 USART串口应用编程步骤

开发一个简单的USART串口通信应用通常需要以下几个步骤：

1. 使能USART时钟

首先，需要使能USART外设的时钟。以USART1为例：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
```

2. 配置GPIO引脚

USART的TX和RX引脚通常是复用引脚，需要配置为相应的复用模式。例如配置PA9和PA10为USART1的TX和RX：

```
GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);  // 使能GPIOA时钟

// 配置TX引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOA, &GPIO_InitStructure);

// 配置RX引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

3. 配置USART参数

通过`USART_Init`函数配置USART的波特率、数据位、停止位和校验位等。

```
USART_InitTypeDef USART_InitStructure;
USART_InitStructure.USART_BaudRate = 9600;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
USART_Init(USART1, &USART_InitStructure);
```

4. 启用USART

配置完成后，启用USART模块，使其开始工作：

```
USART_Cmd(USART1, ENABLE);
```

5. 发送和接收数据

通过`USART_SendData`发送数据，通过`USART_ReceiveData`接收数据。

```
// 发送数据
USART_SendData(USART1, 'A');
while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);

// 接收数据
if (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET) {
    uint8_t receivedData = USART_ReceiveData(USART1);
}
```

6. 启用USART中断（可选）

可以配置USART接收中断，使得每当接收到数据时，触发中断服务程序（ISR）进行处理：

```
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
```

在中断服务程序中接收数据：

```
void USART1_IRQHandler(void) {
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) {
        uint8_t receivedByte = USART_ReceiveData(USART1);
        USART_ClearITPendingBit(USART1, USART_IT_RXNE); // 清除中断标志
    }
}
```

7.4.3 USART标准外设库应用实例

下面是一个基于STM32的简单USART应用实例，其中包含了USART的基本配置、数据发送和接收功能。

应用场景

通过USART1与PC进行串口通信，当接收到数据时，LED灯的状态会发生变化。

代码实现

```
#include "stm32f10x.h"

void GPIO_Config(void) {
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // 配置TX引脚 (PA9)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 配置RX引脚 (PA10)
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
}

void USART_Config(void) {
    USART_InitTypeDef USART_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);

    // 配置USART
    USART_InitStructure.USART_BaudRate = 9600;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
    USART_Init(USART1, &USART_InitStructure);

    // 启用USART1
    USART_Cmd(USART1, ENABLE);
}
```

```
void USART_SendChar(uint8_t c) {
    USART_SendData(USART1, c);
    while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
}

uint8_t USART_ReceiveChar(void) {
    while (USART_GetFlagStatus(USART1, USART_FLAG_RXNE) == RESET);
    return USART_ReceiveData(USART1);
}

int main(void) {
    GPIO
```

_Config(); // 配置GPIO引脚USART_Config(); // 配置USART

```
while (1) {
    uint8_t receivedData = USART_ReceiveChar(); // 接收数据
    if (receivedData == '1') {
        GPIO_SetBits(GPIOA, GPIO_Pin_1); // 点亮LED
    } else if (receivedData == '0') {
        GPIO_ResetBits(GPIOA, GPIO_Pin_1); // 熄灭LED
    }
    USART_SendChar(receivedData); // 回显接收到的数据
}
```

}

总结

STM32的USART模块通过标准外设库提供了丰富的接口函数，简化了串口通信的配置与应用开发。通过合理配置USART的波特率、数据格式、停止位等参数，可以实现可靠的串口通信应用。结合中断和DMA功能，可以进一步优化应用的性能和响应能力。

第8章 DMA

8.1 DMA基础理论知识

DMA（Direct Memory Access，直接存储器访问）是一种允许外设直接与内存交换数据的技术，无需CPU干预。通过DMA，可以在不占用CPU资源的情况下，实现高效的内存数据传输，显著提高数据传输速度，降低CPU的负担。DMA常用于需要高速传输大数据量的场景，如音视频处理、传感器数据采集等。

1. DMA的工作原理

- DMA允许外设或内存之间的数据直接传输，无需CPU介入。
- 通过DMA控制器（DMA Controller）进行数据的传输，CPU负责配置DMA控制器。
- 传输过程中，DMA控制器会根据预设的传输条件（如数据量、源地址、目标地址等）进行数据的传送。

2. DMA的优势

- **减轻CPU负担**：由于数据传输不需要CPU干预，CPU可以执行其他任务。

- **提高数据传输速度**：DMA能够在更短时间内完成数据传输，特别适合高速数据传输。
- **降低能耗**：通过减少CPU工作负担，DMA可以降低系统的能耗。

3. DMA的主要组成

- **DMA控制器**：负责管理DMA传输。
- **通道**：DMA控制器有多个通道，每个通道可以处理一个数据传输任务。
- **源地址和目标地址**：配置数据传输的起始地址和目标地址。
- **数据长度**：配置传输的数据量。
- **传输方向**：配置数据的传输方向（内存到外设，外设到内存，内存到内存等）。

4. DMA传输类型

- **内存到外设**（Memory to Peripheral）
- **外设到内存**（Peripheral to Memory）
- **内存到内存**（Memory to Memory）

8.2 STM32的DMA模块

STM32微控制器的DMA模块与外设和内存之间的数据传输密切相关，支持多种模式和功能，能够显著提高数据传输效率。在STM32中，DMA控制器通常与多个外设（如USART、SPI、I2C、ADC等）结合使用，实现高速的数据交换。

8.2.1 DMA内部结构

STM32的DMA模块通常包含多个DMA通道，每个通道可以配置为与不同的外设或内存进行数据传输。DMA控制器的内部结构包括以下几个关键部分：

1. **DMA控制器**：
   - STM32的DMA控制器负责管理DMA的所有传输任务，并根据配置的传输类型进行数据传送。
   - 每个DMA控制器支持多个DMA通道。每个通道负责处理一对源地址和目标地址之间的数据传输。
2. **DMA通道**：
   - 每个DMA通道可以与一个外设（例如USART、ADC等）或内存进行数据传输。
   - 每个通道可以配置为内存到外设、外设到内存、内存到内存等不同传输方向。
   - 每个通道支持不同的传输模式，如单次传输、循环传输等。
3. **DMA流**：
   - 在某些STM32系列（例如STM32F4）中，DMA通道可以进一步划分为多个流，流和通道的配置可以分别管理不同的传输任务。
4. **配置寄存器**：
   - DMA控制器包含多个配置寄存器，允许开发者配置传输的数据量、源地址、目标地址、数据宽度等参数。
   - 例如，DMA的传输控制寄存器（DMA_CCR）控制着数据传输的各个方面，如传输方向、传输模式等。
5. **DMA状态寄存器**：
   - DMA控制器还包含状态寄存器，用于监控DMA传输的状态，标志传输是否完成或是否发生错误。

8.2.2 DMA优先权

DMA控制器支持配置不同DMA通道的优先权，以确定在多个DMA通道同时请求数据传输时，哪些通道获得优先服务。DMA优先权配置对于确保重要外设的数据传输及时性至关重要。

STM32的DMA优先级通常有两种级别：高优先级和低优先级。通过配置DMA的优先级寄存器，可以为每个DMA通道分配不同的优先级。

优先级设置有两种方式：

1. **固定优先级**：DMA控制器按照优先级顺序处理多个通道的请求，较高优先级的通道优先进行数据传输。
2. **轮询优先级**：DMA控制器按顺序处理各个通道的请求，没有固定的优先级。每次DMA通道请求时，系统会轮流响应。

优先级的设置可以影响系统中多个DMA请求的响应顺序。例如，当多个外设通过DMA进行数据传输时，优先级较高的外设可以获得更快的传输速度。

8.2.3 DMA中断请求

在DMA数据传输过程中，可以使用中断来通知CPU传输的状态。例如，当传输完成时，DMA控制器会触发中断请求，CPU可以根据中断信号进行后续处理。DMA模块支持多种中断请求，包括：

1. **传输完成中断（TC）**：
   - 当DMA传输完成时，DMA控制器会触发传输完成中断。开发者可以在中断服务程序中进行后续处理。
2. **传输错误中断（TE）**：
   - 当DMA传输发生错误时，DMA控制器会触发传输错误中断。错误可能包括地址溢出、数据位错误等。
3. **半传输中断（HT）**：
   - 当DMA传输到一半时，触发半传输中断。可以用于实时数据处理应用，如音频处理、视频处理等。
4. **FIFO溢出中断（FO）**：
   - 当DMA的FIFO（先进先出）缓冲区溢出时，触发FIFO溢出中断。FIFO溢出通常发生在数据流速过快的情况下，DMA缓冲区未能及时处理数据。

配置DMA中断：

1. **使能DMA中断**：
   - 使用`DMA_ITConfig()`函数使能DMA的相关中断：

```
DMA_ITConfig(DMA1_Channel1, DMA_IT_TC, ENABLE);  // 启用传输完成中断
```

2. **中断服务程序（ISR）**：
   - 在中断服务程序中，可以处理DMA传输完成或错误等事件：

```
void DMA1_Channel1_IRQHandler(void) {
    if (DMA_GetITStatus(DMA1_IT_TC1)) { // 判断是否为传输完成中断
        DMA_ClearITPendingBit(DMA1_IT_TC1); // 清除中断标志
        // 处理DMA传输完成事件
    }
}
```

3. **清除中断标志**：
   - 在中断处理函数中，需要清除DMA中断标志，以确保系统可以响应后续的中断请求：

```
DMA_ClearITPendingBit(DMA1_IT_TC1);  // 清除传输完成中断标志
```

通过使用DMA中断，开发者可以在传输完成时获得通知，避免CPU不断轮询DMA状态，从而节省系统资源。

总结

DMA（直接存储器访问）是STM32微控制器中用于高效数据传输的关键功能，能够实现外设与内存之间的数据交换而无需占用CPU时间。通过合理配置DMA的优先级、传输方向和中断请求，可以优化系统的数据传输效率和响应速度。DMA广泛应用于音频处理、图像处理、数据采集等需要高效数据传输的场景。STM32的DMA模块具有强大的灵活性和配置选项，可以满足不同的应用需求。

8.3 DMA标准外设库接口函数及应用

8.3.1 DMA标准外设库接口函数

DMA（Direct Memory Access，直接内存存取）是用于在外设和内存之间直接传输数据的一种机制，能够有效减少CPU的干预，提高数据传输效率。在STM32等嵌入式平台中，DMA通常通过标准外设库来配置和管理。以下是DMA标准外设库的主要接口函数。

1. DMA初始化函数

- *DMA_Init(DMA_Channel_TypeDef DMA_Channel, DMA_InitTypeDef* DMA_InitStruct)** 用于初始化DMA通道的配置。主要配置源地址、目标地址、数据传输方向、数据大小等。
- **参数说明：**
  - `DMA_Channel`：需要初始化的DMA通道（如DMA1_Channel1）。
  - `DMA_InitStruct`：配置结构体，包含了DMA的配置项（传输方向、数据大小、循环模式等）。

2. DMA配置结构体

- DMA_InitTypeDef

  配置DMA通道的结构体，包含以下主要字段：
  - `DMA_PeripheralBaseAddr`：外设基地址。
  - `DMA_MemoryBaseAddr`：内存基地址。
  - `DMA_DIR`：数据传输方向（从外设到内存，或从内存到外设）。
  - `DMA_BufferSize`：数据传输的字节数。
  - `DMA_PeripheralDataSize`：外设数据宽度。
  - `DMA_MemoryDataSize`：内存数据宽度。
  - `DMA_Mode`：传输模式（正常模式、循环模式）。
  - `DMA_Priority`：DMA传输优先级。
  - `DMA_FIFOMode`：FIFO模式使能。

3. DMA启动函数

- *DMA_Cmd(DMA_Channel_TypeDef DMA_Channel, FunctionalState NewState)** 启动或停止DMA传输。
- **参数说明：**
  - `DMA_Channel`：DMA通道。

- `NewState`：启动DMA传输（`ENABLE`）或停止（`DISABLE`）。

4. DMA中断配置函数

- *DMA_ITConfig(DMA_Channel_TypeDef DMA_Channel, DMA_IT_TypeDef DMA_IT, FunctionalState NewState)** 配置DMA中断，设置DMA传输完成或错误时触发中断。
- **参数说明：**
  - `DMA_Channel`：DMA通道。
  - `DMA_IT`：DMA中断类型，如传输完成中断、传输错误中断等。
  - `NewState`：启用（`ENABLE`）或禁用（`DISABLE`）中断。

5. DMA标志检查与清除

- *DMA_GetFlagStatus(DMA_Channel_TypeDef DMA_Channel, uint32_t DMA_FLAG)**
  检查DMA标志，确定DMA传输是否完成。
- *DMA_ClearFlag(DMA_Channel_TypeDef DMA_Channel, uint32_t DMA_FLAG)**
  清除DMA传输完成的标志。
- *DMA_GetITStatus(DMA_Channel_TypeDef DMA_Channel, DMA_IT_TypeDef DMA_IT)**
  检查DMA中断状态。
- *DMA_ClearITPendingBit(DMA_Channel_TypeDef DMA_Channel, DMA_IT_TypeDef DMA_IT)**
  清除DMA中断挂起位。

8.3.2 DMA标准外设库应用编程步骤

使用DMA时，一般需要按照以下步骤进行编程：

1. **配置DMA通道**：
   - 首先，初始化DMA通道的配置结构体（`DMA_InitTypeDef`），设置数据源、数据目标、数据大小等。
   - 调用`DMA_Init()`函数将配置应用到指定的DMA通道。
2. **配置外设**：
   - 配置外设（例如ADC、SPI、USART等）进行DMA传输。外设通常需要启用DMA支持，并配置相应的DMA请求。
3. **启用DMA**：
   - 调用`DMA_Cmd()`函数启动DMA传输。
4. **配置DMA中断（可选）**：
   - 如果需要在DMA传输完成后进行处理，可以配置DMA中断。启用相应的中断，设置中断服务程序。
5. **等待DMA传输完成**：
   - 使用`DMA_GetFlagStatus()`检查DMA传输是否完成，或在中断服务程序中处理传输完成事件。
6. **清除DMA标志和中断**：
   - DMA传输完成后，需要清除DMA标志和中断挂起位，准备下一次传输。

8.3.3 DMA标准外设库应用实例

以下是一个简单的DMA应用实例，演示如何使用DMA将数据从内存传输到外设。

示例：使用DMA将数据从内存传输到USART

1. **初始化DMA通道**：

```
DMA_InitTypeDef DMA_InitStructure;
DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&USART1->DR;  // 外设基地址
DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)txBuffer;         // 内存基地址
DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToPeripheral;             // 数据传输方向：内存到外设
DMA_InitStructure.DMA_BufferSize = sizeof(txBuffer);               // 传输数据的字节数
DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;
DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;
DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;                       // 正常模式
DMA_InitStructure.DMA_Priority = DMA_Priority_High;                 // 高优先级
DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;              // 禁用FIFO
DMA_Init(DMA1_Channel4, &DMA_InitStructure);  // 初始化DMA通道
```

1. **配置USART外设以支持DMA传输**：

```
USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE);  // 启用USART的DMA传输请求
```

1. **启动DMA传输**：

```
DMA_Cmd(DMA1_Channel4, ENABLE);  // 启动DMA传输
```

1. **等待传输完成**：

```
while (DMA_GetFlagStatus(DMA1_FLAG_TC4) == RESET);  // 等待传输完成
```

1. **清除DMA标志**：

```
DMA_ClearFlag(DMA1_FLAG_TC4);  // 清除传输完成标志
```

在这个例子中，数据通过DMA从 `txBuffer` 传输到USART1的数据寄存器。当传输完成时，DMA会自动停止传输。

以上是一个简单的DMA应用，实际应用中，DMA通常用于更复杂的操作，如数据采集、音频传输等。

第9章 定时器

定时器是嵌入式系统中非常重要的外设之一，广泛应用于产生延时、生成PWM信号、测量时间间隔等。STM32的定时器模块具有多个功能，可以根据需要配置为不同的定时器类型。

9.1 STM32定时器模块

STM32系列微控制器提供了多种类型的定时器，包括通用定时器、基本定时器和高级定时器。每种定时器都有其独特的功能，适用于不同的应用场景。

9.1.1 通用定时器

通用定时器（General-purpose timers）是STM32中最常用的一类定时器，适用于一般的定时、计数、PWM输出等应用。

功能特点：

- **计数模式**：可以配置为向上计数、向下计数或向上/向下计数。

- **PWM输出**：可以用于生成PWM信号，用于控制电机、LED调光等。
- **输入捕获**：可以用于测量信号的周期、频率等参数。
- **输出比较**：可以根据设定的阈值产生定时中断，进行某些任务的触发。
- **死区时间控制**：可用于双通道的PWM输出，控制通道之间的死区时间。

常见的通用定时器：

- **TIM2、TIM3、TIM4、TIM5** 等，这些定时器可以在各种模式下工作，如普通定时模式、PWM模式、输入捕获模式等。

典型应用：

- 定时器1和定时器2常用于产生PWM信号控制电机或LED亮度。
- 定时器3和定时器4可以用作输入捕获模式，测量外部信号的频率。

配置实例（PWM输出）：

```
// 1. 初始化定时器
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 999;  // 设置PWM周期
TIM_TimeBaseStructure.TIM_Prescaler = 71;  // 设置预分频器
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);

// 2. 配置PWM输出模式
TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = 499;  // 设置占空比
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OC1Init(TIM2, &TIM_OCInitStructure);

// 3. 启动定时器
TIM_Cmd(TIM2, ENABLE);
TIM_CtrlPWMOutputs(TIM2, ENABLE);
```


9.1.2 基本定时器

基本定时器（Basic timers）通常用于较简单的定时和计数任务。它们没有PWM功能，也没有输入捕获和输出比较功能，主要用于产生定时中断。

功能特点：

- **定时功能**：最简单的计时器，主要用于生成定时中断。
- **简化的配置**：不支持PWM输出、输入捕获等复杂功能。
- **单通道模式**：没有高级定时器或通用定时器那样的多个通道支持。

常见的基本定时器：

- **TIM6、TIM7** 等，这些定时器只能提供最基本的定时功能，通常用于产生固定时间间隔的中断。

典型应用：

- 用于延时生成、系统节拍（SysTick）等。
- 在某些低功耗应用中，基本定时器也可以在低功耗模式下运行，节省能源。

配置实例（定时中断）：

```
// 1. 配置基本定时器
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 999;  // 设置定时器周期
TIM_TimeBaseStructure.TIM_Prescaler = 71;  // 设置定时器预分频器
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM6, &TIM_TimeBaseStructure);

// 2. 启动定时器
TIM_Cmd(TIM6, ENABLE);
```

9.1.3 高级定时器

高级定时器（Advanced-control timers）主要用于需要精确控制和多个输出通道的应用。它们提供了更加复杂的功能，适用于电机控制、复杂PWM生成等应用。

功能特点：

- **高精度PWM输出**：支持多个输出通道，适用于电机控制等高精度场景。
- **死区时间控制**：支持通道之间的死区时间，避免输出信号之间的短路。
- **输入捕获与输出比较**：支持高级的输入捕获与输出比较功能。
- **多通道PWM输出**：支持生成多个通道的PWM信号。
- **反向PWM**：可以控制PWM信号的反向输出。

常见的高级定时器：

- **TIM1、TIM8** 等，这些定时器具有多种高级功能，包括死区时间控制、双通道输出比较等。

典型应用：

- 电机控制：使用多通道PWM输出。
- 精密计时：应用于需要多个输出信号控制的系统。
- 高精度PWM和双通道输出，应用于高功率设备控制。

配置实例（双通道PWM输出）：

```
// 1. 配置高级定时器
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 999;  // 设置PWM周期
TIM_TimeBaseStructure.TIM_Prescaler = 71;  // 设置预分频器
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);

// 2. 配置PWM输出
TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
```

```
TIM_OCInitStructure.TIM_Pulse = 499;  // 设置占空比
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;
TIM_OC1Init(TIM1, &TIM_OCInitStructure);
TIM_OC2Init(TIM1, &TIM_OCInitStructure);

// 3. 启动定时器
TIM_Cmd(TIM1, ENABLE);
TIM_CtrlPWMOutputs(TIM1, ENABLE);
```

总结：

- **通用定时器**：适用于一般计时、PWM输出、输入捕获和输出比较。
- **基本定时器**：功能简单，适用于仅需要定时中断的应用。
- **高级定时器**：具有复杂功能，适用于电机控制、精确PWM输出等高精度控制的场景。

9.2 定时器标准外设库接口函数及应用

STM32的定时器模块提供了丰富的功能，通过标准外设库接口函数可以轻松地进行配置和控制。以下是关于定时器的标准外设库接口函数、应用编程步骤和实例。

9.2.1 定时器标准外设库接口函数

定时器的标准外设库函数主要用于定时器的初始化、启动、配置模式、生成PWM输出、设置中断等功能。以下是定时器相关的标准外设库接口函数。

1. 定时器初始化函数

- **TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct)** 用于初始化定时器的基本配置，如计数模式、预分频、自动重载值（周期）等。
- **参数说明：**
  - `TIMx`：指定的定时器（如 `TIM1`，`TIM2` 等）。
  - `TIM_TimeBaseInitStruct`：配置结构体，包含计数模式、预分频、自动重载值等。
- **配置结构体 TIM_TimeBaseInitTypeDef**：
  - `TIM_Period`：自动重载值，控制定时器计数周期。
  - `TIM_Prescaler`：预分频器，设置时钟频率。
  - `TIM_ClockDivision`：时钟分频，设置计数时钟的分频系数。
  - `TIM_CounterMode`：计数模式（向上计数、向下计数、向上/向下计数）。
  - `TIM_RepetitionCounter`：重复计数器，用于高级定时器的死区时间控制。

2. 定时器输出比较和PWM配置

- **TIM_OCInit(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct)** 配置定时器的输出比较模式，可以用于PWM输出、比较事件等。
- **参数说明：**
  - `TIMx`：指定的定时器（如 `TIM1`，`TIM2` 等）。
  - `TIM_OCInitStruct`：配置输出比较的结构体，包含工作模式、占空比等。
- **配置结构体 TIM_OCInitTypeDef**：
  - `TIM_OCMode`：输出比较模式（如PWM模式、正向/反向PWM）。
  - `TIM_OutputState`：输出状态（使能或禁用）。

- `TIM_Pulse`：输出比较值，用于控制PWM的占空比。
- `TIM_OCPolarity`：输出极性（高电平或低电平）。

3. 定时器中断配置函数

- **TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState)**
  配置定时器中断，使能或禁用特定的中断源（如更新中断、捕获中断等）。
- **TIM_ClearFlag(TIM_TypeDef* TIMx, uint16_t TIM_FLAG)**
  清除定时器的标志位，通常用于清除定时器溢出等标志。
- **TIM_ClearITPendingBit(TIM_TypeDef* TIMx, uint16_t TIM_IT)**
  清除定时器的中断挂起位，通常在中断服务函数中使用。

4. 定时器启动和停止

- **TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState)**
  启动或停止指定的定时器。
- **TIM_CtrlPWMOutputs(TIM_TypeDef* TIMx, FunctionalState NewState)**
  启用或禁用定时器的PWM输出，适用于高级定时器和通用定时器。

9.2.2 定时器标准外设库应用编程步骤

使用STM32定时器时，应用编程通常按照以下步骤进行：

1. **配置定时器基础参数**：
   - 选择合适的定时器（如`TIM1`、`TIM2`等）。
   - 配置定时器的计数模式（向上计数、向下计数等）、预分频器、自动重载值等。
   - 调用`TIM_TimeBaseInit()`函数初始化定时器基础配置。
2. **配置定时器输出比较或PWM**：
   - 如果需要PWM输出，配置定时器的输出比较模式，设置占空比等参数。
   - 调用`TIM_OCInit()`函数配置PWM或输出比较模式。
3. **配置定时器中断（如果需要）**：
   - 如果需要在定时器溢出或其他事件发生时进行中断处理，配置定时器中断源。
   - 使用`TIM_ITConfig()`使能中断，配置中断优先级和中断服务函数。
4. **启动定时器**：
   - 调用`TIM_Cmd()`启动定时器。
   - 对于PWM输出，使用`TIM_CtrlPWMOutputs()`使能PWM输出。
5. **处理中断（如果需要）**：
   - 在中断服务函数中处理定时器中断，清除中断标志或执行其他操作。
6. **清除标志位（如果需要）**：
   - 在中断服务程序或主程序中使用`TIM_ClearFlag()`和`TIM_ClearITPendingBit()`清除标志位和中断挂起位。

9.2.3 定时器标准外设库应用实例

以下是一个典型的应用实例，展示如何使用定时器产生PWM信号，控制一个LED的亮度。

示例：使用TIM2生成PWM信号控制LED亮度

1. **初始化定时器2**（TIM2）：

```
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 999;  // PWM周期（频率）
TIM_TimeBaseStructure.TIM_Prescaler = 71;  // 预分频器（定时器频率）
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
```

1. **配置PWM输出通道1**：

```
TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;  // PWM模式
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = 499;  // 占空比（50%）
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;  // 正极性
TIM_OC1Init(TIM2, &TIM_OCInitStructure);
```

1. **启动定时器2**：

```
TIM_Cmd(TIM2, ENABLE);  // 启动定时器
TIM_CtrlPWMOutputs(TIM2, ENABLE);  // 启用PWM输出
```

1. **改变占空比**（动态调整亮度）：

```
// 修改占空比为75%
TIM_SetCompare1(TIM2, 749);  // 占空比 = 749 / 999
```

1. **清除中断标志（如果需要）**：

```
if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET) {
    // 处理溢出事件
    TIM_ClearITPendingBit(TIM2, TIM_IT_Update);  // 清除更新中断标志
}
```


解释：

- **PWM输出**：定时器2生成一个PWM信号，控制LED的亮度。通过调整`TIM_OCInitStructure.TIM_Pulse`值，可以改变PWM的占空比，从而控制LED的亮度。
- **定时器配置**：通过`TIM_TimeBaseInit()`设置定时器周期（`TIM_Period`）和预分频器（`TIM_Prescaler`），确保定时器的频率符合应用要求。
- **动态调整**：通过`TIM_SetCompare1()`动态修改占空比，改变LED的亮度。

总结：

- **定时器的应用编程**包括初始化、配置PWM或输出比较模式、配置中断、启动定时器等步骤。
- 使用定时器进行PWM输出控制是非常常见的应用，尤其在电机控制、LED调光等场景中。
- 通过定时器的标准外设库接口函数，可以轻松实现各种定时功能，提供高效和精确的定时控制。

9.4 PWM（脉宽调制）

PWM（Pulse Width Modulation，脉宽调制）是一种通过调节信号的高电平持续时间（即占空比）来控制功率输出的技术。它广泛应用于电机控制、亮度调节、音频信号生成等多个领域。STM32微控制器通过其定时器模块提供了丰富的PWM输出功能，可以通过标准外设库（Standard Peripheral Library）或HAL库进行配置和使用。

9.4.1 PWM的工作原理

PWM的基本原理是通过调节周期内信号高电平（ON）的时间占整个周期的比例来控制输出功率。周期内的高电平时间越长，占空比（Duty Cycle）就越大，输出的平均功率也就越大。

- **周期（Period）**：PWM信号的周期是指信号从一个完整周期的开始到下一个周期的开始所经过的时间。周期通常用秒（s）来表示。
- **占空比（Duty Cycle）**：占空比是高电平的持续时间与整个周期时间的比值，通常用百分比表示。例如，占空比为50%的PWM信号表示高电平持续时间占整个周期的50%。

例如：

- **占空比 0%**：信号一直处于低电平。
- **占空比 50%**：高电平和低电平各占周期的一半。
- **占空比 100%**：信号一直处于高电平。

PWM的输出信号：

- **频率（Frequency）**：频率是周期的倒数，单位为赫兹（Hz）。它决定了PWM信号切换的速度。
- **占空比（Duty Cycle）**：占空比越大，信号的平均功率越高。

9.4.2 PWM的标准外设库应用实例

在STM32中，使用标准外设库来实现PWM输出的步骤通常包括定时器的初始化、配置PWM输出模式、设置占空比等。以下是一个简单的PWM输出实例，使用STM32的定时器（例如TIM2）来产生PWM信号。

1. 初始化定时器2（作为PWM生成器）

```
// 定时器2的基础配置
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
TIM_TimeBaseStructure.TIM_Period = 999;  // 设置PWM周期，周期为1000个时钟周期
TIM_TimeBaseStructure.TIM_Prescaler = 71;  // 设置预分频器，分频系数为72（定时器时钟为72 MHz）
TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;  // 向上计数模式
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
```

2. 配置PWM输出通道1（使用TIM2的通道1生成PWM）

```
// 配置定时器2的通道1为PWM模式
TIM_OCInitTypeDef TIM_OCInitStructure;
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;  // PWM模式
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = 499;  // 设置PWM的占空比为50%（499/999）
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;  // 正极性
TIM_OC1Init(TIM2, &TIM_OCInitStructure);
```

3. 启动定时器2并使能PWM输出

```
// 启动定时器2
TIM_Cmd(TIM2, ENABLE);
// 启用PWM输出
TIM_CtrlPWMOutputs(TIM2, ENABLE);
```

4. 动态调整PWM占空比

```
// 设置占空比为75% (750/999)
TIM_SetCompare1(TIM2, 750);  // 修改比较值以调节占空比
```

9.4.3 PWM的HAL库应用实例

HAL库（硬件抽象层）提供了更加简洁的接口来配置和控制硬件外设。使用HAL库配置PWM的过程与标准外设库类似，但接口更简单、易于理解。以下是一个使用HAL库产生PWM信号的例子，假设使用TIM2通道1来生成PWM信号。

1. 初始化定时器2（作为PWM生成器）

```
// 定义定时器的句柄
TIM_HandleTypeDef htim2;

// 初始化定时器2
htim2.Instance = TIM2;
htim2.Init.Prescaler = 71;  // 设置预分频器
htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
htim2.Init.Period = 999;  // 设置PWM周期
htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_PWM_Init(&htim2) != HAL_OK) {
    // 错误处理
}
```

2. 配置PWM输出通道1（使用TIM2的通道1生成PWM）

```
// 配置PWM的参数
TIM_OC_InitTypeDef sConfigOC;
sConfigOC.OCMode = TIM_OCMODE_PWM1;  // PWM模式
sConfigOC.Pulse = 499;  // 占空比50% (499/999)
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;  // 正极性
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK) {
    // 错误处理
}
```

3. 启动PWM输出

```
// 启动PWM输出
if (HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1) != HAL_OK) {
    // 错误处理
}
```

4. 动态调整PWM占空比

```
// 设置占空比为75% (750/999)
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 750);  // 修改比较值以调节占空比
```

解释：

- **初始化定时器**：首先通过 `HAL_TIM_PWM_Init()` 初始化定时器，并配置时钟、预分频器、自动重载值等。
- **配置PWM通道**：通过 `HAL_TIM_PWM_ConfigChannel()` 设置PWM模式、占空比、极性等参数。
- **启动PWM输出**：使用 `HAL_TIM_PWM_Start()` 启动PWM输出。
- **动态调整占空比**：使用 `__HAL_TIM_SET_COMPARE()` 动态调整PWM占空比，从而改变输出信号的高电平持续时间。

总结：

- **PWM的工作原理**是通过改变占空比来调节输出的平均功率，在很多控制应用中，如电机驱动、LED调光等，PWM技术是非常重要的。
- **标准外设库**和**HAL库**提供了简单的接口来配置和使用PWM。标准外设库功能较为底层，提供更多控制选项，而HAL库则对外设进行了更高层次的封装，简化了开发过程。
- 通过定时器模块，STM32能够非常方便地实现PWM信号的产生，并且可以动态调整PWM信号的占空比，满足不同的应用需求。

9.5 SysTick定时器

SysTick定时器是STM32微控制器中的一个内置定时器，通常用于实现系统时基、产生定时中断或用于调度任务。SysTick定时器是一个24位的递增定时器，通常用于操作系统的时间管理、产生节拍信号等。SysTick定时器的时钟来源通常是系统时钟（如HCLK或MCLK），并且它能够产生定时中断。

9.5.1 SysTick标准外设库函数

SysTick定时器的标准外设库提供了几个重要的函数，用于配置和控制SysTick定时器。以下是常用的标准外设库函数：

1. **SysTick_Config(uint32_t Ticks)**
   - 配置SysTick定时器。
   - 参数说明

     ：
     - `Ticks`：定时器计数的初始值，表示SysTick定时器的重载值，即定时器中断的周期。该值通常是系统时钟频率与需要的延迟时间的商。
   - **功能**：配置SysTick定时器，使其根据系统时钟周期自动计数，并在计数达到指定的 `Ticks` 值时触发中断。
2. **SysTick->CTRL**
   - 控制SysTick定时器的启用、定时源和中断使能。可以设置SysTick的不同工作模式。
   - 通过修改SysTick的控制寄存器 `CTRL`，可以使能/禁用SysTick定时器、选择时钟源（HCLK或内部时钟）和使能中断。
3. **SysTick->LOAD**
   - 设置SysTick定时器的重载值，即计数周期。

4. **SysTick->VAL**
    - 读取SysTick定时器当前的计数值。
5. **SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk**
    - 检查SysTick定时器是否已经完成一个周期，返回计数标志。
6. **SysTick_CLKSourceConfig(SysTick_CLKSource)**
    - 配置SysTick时钟源，可以选择内部时钟或外部时钟。

9.5.2 SysTick标准外设库应用实例

SysTick定时器的常见应用包括实现延时函数、定时器中断生成等。以下是一个使用SysTick定时器来生成定时中断的例子。

示例：使用SysTick实现定时中断

1. **配置SysTick定时器并启动**

```
// 配置SysTick定时器以每1ms触发一次中断
SysTick_Config(SystemCoreClock / 1000);  // 系统时钟频率除以1000，得到1ms的定时周期
```

1. **SysTick中断服务程序**

```
// SysTick中断处理函数
void SysTick_Handler(void)
{
    // 每次SysTick中断发生时执行此函数
    // 例如，增加1ms的计数，或执行其他周期性任务
    static uint32_t msCounter = 0;
    msCounter++;

    // 每1000ms（1秒）时，切换LED状态
    if (msCounter >= 1000) {
        msCounter = 0;
        // 切换LED状态
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0);
    }
}
```

1. **中断优先级设置（可选）**

在一些情况下，可能需要设置SysTick中断的优先级。例如，在使用FreeRTOS时，可能需要调整中断的优先级来与RTOS的任务调度配合。

```
// 设置SysTick中断的优先级（如果需要）
NVIC_SetPriority(SysTick_IRQn, 1);  // 设置SysTick中断的优先级为1（较低）
```

1. **清除SysTick中断标志**

如果需要手动清除SysTick中断标志（通常在中断服务函数中），可以通过读取`SysTick->CTRL`寄存器来清除中断标志。

```
// 清除SysTick中断标志
if (SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk) {
    // 执行相关操作
}
```

9.6 编程思想之状态机设计思想

状态机（State Machine）是一种常用于嵌入式系统开发中的编程思想，尤其适用于处理多种不同状态和事件的应用。状态机通过一组状态及其转换规则来控制程序的执行流，通常用于任务调度、协议解析、设备控制等应用场景。

状态机的基本概念

1. **状态（State）**：表示系统的某种特定情况或条件。每个状态可能有特定的行为、输出或执行某个操作。
2. **事件（Event）**：引起状态变化的触发因素，事件可以是外部输入（如传感器数据、按键输入）或系统内部的条件（如计时器溢出）。
3. **状态转换（Transition）**：根据某个事件，系统从一个状态转移到另一个状态。每个状态之间的转换有一定的条件和动作。
4. **状态机表**：通常使用一个状态转换表来记录每个状态与事件的对应关系。这张表定义了在某个特定状态下，遇到特定事件时应该转移到哪个状态，并且可能会执行某个特定的动作。

状态机的设计步骤

1. **确定状态**：首先，明确系统的所有可能状态，并为每个状态定义一个唯一的标识符。
2. **确定事件**：定义所有可能引起状态变化的事件。例如，按钮按下、传感器值变化、计时器到期等。
3. **确定状态转换规则**：为每个状态定义在不同事件下的状态转换规则，并明确转换条件和执行的动作。
4. **实现状态机逻辑**：编写代码来表示状态机的状态、事件、转换规则和动作。通常使用`switch-case`或`if-else`语句来实现状态的切换和事件的处理。

示例：简单的LED控制状态机

假设我们设计一个LED控制系统，LED有两种状态：开和关。系统通过按键输入来切换LED的状态。

1. **定义状态**：

```
typedef enum {
    LED_OFF = 0,
    LED_ON
} LED_State;
```

1. **定义事件**：

```
typedef enum {
    BUTTON_PRESSED = 0
} Event;
```

1. **定义状态机**：

```
LED_State currentState = LED_OFF;  // 初始状态为LED关

void LED_StateMachine(Event event) {
```

```
    switch (currentState) {
        case LED_OFF:
            if (event == BUTTON_PRESSED) {
                currentState = LED_ON;  // 按钮按下时切换到LED开
                HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);  // 打开LED
            }
            break;
        case LED_ON:
            if (event == BUTTON_PRESSED) {
                currentState = LED_OFF;  // 按钮按下时切换到LED关
                HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);  // 关闭LED
            }
            break;
        default:
            break;
    }
}
```

1. **调用状态机**：

```
// 在主循环或定时中断中，根据按键输入调用状态机
if (buttonPressed) {
    LED_StateMachine(BUTTON_PRESSED);
}
```

状态机的优势

1. **清晰的控制逻辑**：状态机将复杂的控制逻辑分解为状态和事件，使得程序更易于理解和维护。
2. **灵活性**：状态机可以轻松地扩展新的状态和事件。例如，如果需要增加一个新的LED闪烁状态，只需添加新的状态并定义相应的事件处理规则。
3. **可测试性**：状态机的每个状态和事件可以独立地进行测试，确保程序逻辑的正确性。
4. **响应性**：状态机能够快速响应外部事件（如按键、传感器变化等），并根据状态转换规则执行适当的操作。

总结

- **SysTick定时器**是一个重要的硬件定时器，常用于产生系统时基、实现延时或定时任务。通过配置SysTick定时器和编写相应的中断服务程序，可以实现精确的时间控制。
- **状态机设计思想**是嵌入式系统中常用的一种编程方法，它通过明确定义的状态、事件和状态转换规则来控制程序的执行流程。状态机简化了程序设计，特别适合处理多状态、多事件的应用，如任务调度、设备控制等。

第10章 模拟数字转换（ADC）

模拟数字转换（Analog-to-Digital Conversion，简称ADC）是将连续的模拟信号（例如电压）转换为离散的数字信号的过程。STM32系列微控制器内置了高性能的ADC模块，可以用于各种传感器的信号采集、数字信号处理等应用。

10.1 ADC基础理论知识

模拟信号通常是连续的，而数字信号则是离散的。ADC的主要作用是将输入的模拟信号转换为可以由数字处理单元（如微控制器）处理的数字信号。通过采样和量化过程，ADC可以实现这一转换。

10.1.1 A/D转换过程

A/D转换过程可以分为以下几个步骤：

1. **采样（Sampling）**
   - 采样过程是将连续的模拟信号在特定的时间点进行离散化。在此过程中，模拟信号在每个采样时刻被读取并保持在一个固定值。
   - 采样定理：为了避免失真，采样频率必须至少是信号最高频率的两倍（奈奎斯特定理）。这个频率被称为采样率。
2. **保持（Hold）**
   - 在采样之后，模拟信号的值会被保持在一个固定的电压上，直到转换完成。保持电路通常使用采样保持电容来存储模拟信号的电压。
3. **量化（Quantization）**
   - 量化是将模拟信号的幅度转换为一个有限的离散值。量化的精度由ADC的分辨率决定，通常用“位数”来表示。例如，12位ADC的量化范围是从0到212−1=40952^{12}-1 = 4095。
   - 量化误差（或量化噪声）是由于模拟信号的幅度值不能精确地映射到数字值所产生的误差。
4. **编码（Encoding）**
   - 在编码步骤中，量化后的离散值会被转换为二进制数字。这些二进制数字可以被数字系统进一步处理或存储。
5. **数字输出**
   - 最终，ADC的输出是一个数字信号，表示模拟信号在采样时刻的幅度。这个数字信号可以直接用于数字计算或通过接口传输到其他系统。

10.1.2 A/D转换的主要技术参数

在选择和设计ADC时，有几个关键参数需要考虑，这些参数决定了ADC的性能和适用性：

1. **分辨率（Resolution）**
   - 分辨率是ADC可以表示的数字输出的位数，通常以“位”表示。例如，12位ADC的分辨率为212=40962^{12} = 4096个离散值。
   - 分辨率越高，ADC能够捕捉到的信号细节就越多，转换的精度也就越高。
2. **转换速率（Conversion Rate）**
   - 转换速率是ADC每秒钟完成多少次转换的速率，通常以“样本/秒”（SPS）或“赫兹”（Hz）来衡量。
   - 在高转换速率下，ADC能够快速地对输入信号进行采样，适用于高速信号处理应用。但同时，高速采样可能增加功耗。
3. **输入范围（Input Range）**
   - 输入范围定义了ADC能够正确转换的模拟输入信号的电压范围。常见的输入范围包括0到Vref（参考电压）或- Vref/2到Vref/2（对称范围）。
   - 输入范围的选择直接影响到模拟信号的采样精度。
4. **参考电压（Reference Voltage，Vref）**
   - 参考电压是ADC的基准电压，用于确定输入信号的最大可转换电压。ADC的最大数字输出值通常对应于参考电压。
   - 如果输入信号的电压超过参考电压，ADC将无法正确转换该信号。因此，参考电压的选择对ADC的转换范围和精度至关重要。

5. **采样时间（Sampling Time）**
    - 采样时间是ADC用于采样输入模拟信号的时间。采样时间越长，输入信号的稳定性越好，从而提高转换精度。
    - 采样时间与ADC的转换精度密切相关，通常需要根据输入信号的特性进行调整。
6. **转换精度（Conversion Accuracy）**
    - 转换精度是指ADC输出结果与实际模拟信号之间的误差。该误差受到ADC的分辨率、量化噪声以及参考电压精度等因素的影响。
7. **输入阻抗（Input Impedance）**
    - 输入阻抗是指ADC输入端的电阻，影响输入信号源与ADC之间的匹配。如果输入阻抗过高，可能会导致信号衰减或采样误差。
8. **采样保持时间（Sample-and-Hold Time）**
    - 采样保持时间是ADC在采样阶段对输入信号进行采样并将其保持的时间。短的保持时间可能导致输入信号不稳定，从而影响转换精度。
9. **线性度（Linearity）**
    - 线性度表示ADC输出值与输入信号之间的关系是否一致。理想的ADC输出应与输入信号成线性关系，任何偏差都被视为非线性误差。
10. **信号噪声（Signal Noise）**
    - ADC的性能受输入信号噪声的影响。信号噪声会影响ADC的转换精度，因此在设计时需要考虑如何减少噪声影响，如使用滤波器等。

总结

- **ADC的基本过程**：包括采样、保持、量化和编码，最终将模拟信号转换为数字信号。ADC的精度和转换速率是设计时的重要参数。
- **主要技术参数**：如分辨率、转换速率、参考电压、采样时间和输入阻抗等，直接影响ADC的性能和适用范围。在嵌入式系统中，合理配置这些参数非常关键，以满足不同应用的需求。

10.2 STM32的ADC

STM32系列微控制器内置了高性能的ADC模块，支持多种转换模式、引脚选择、数据对齐方式等配置，适用于精密的模拟信号采集。STM32的ADC功能丰富，灵活性高，广泛应用于传感器接口、音频处理、信号测量等嵌入式应用中。

10.2.1 ADC的引脚

STM32的ADC模块通常有多个引脚可供选择用于模拟输入信号。不同的引脚可以映射到不同的ADC通道。ADC引脚分为以下几类：

- **标准输入引脚**：这些引脚支持直接的模拟信号输入，通常是通过GPIO配置为模拟模式（Analog mode）来启用。
- **特殊功能引脚**：某些引脚具有多重功能，除了普通GPIO功能外，还可以用作ADC输入通道。
- **内建传感器引脚**：某些STM32微控制器还提供了内建的温度传感器、电压参考信号等输入，这些可以通过内置的ADC通道进行采集。

例如：

- **STM32F4系列**：引脚如`PA0`，`PA1`，`PA2`等都可以作为ADC输入引脚。

- **STM32L4系列**：支持更多的模拟输入通道，例如`PA0`到`PA15`等引脚。

要选择特定的ADC引脚，通常需要在初始化过程中通过STM32的GPIO配置功能将相应引脚设置为模拟模式。

```
// 配置PA0为模拟输入
GPIO_InitTypeDef GPIO_InitStruct = {0};
GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

10.2.2 ADC通道选择

STM32的ADC模块支持多个通道，可以从不同的引脚或内部信号（如温度传感器）读取模拟信号。每个通道都可以独立配置。

- **通道选择**：在STM32的ADC模块中，每个引脚通常都有一个与之对应的ADC通道。通道的选择是通过设置ADC通道的编号来实现的。
- **通道顺序**：多个通道可以按顺序依次转换，ADC的通道扫描模式（扫描模式）允许一次启动多个通道的转换。

例如，STM32F4系列的ADC通道选择：

- `ADC_CHANNEL_0`对应引脚`PA0`
- `ADC_CHANNEL_1`对应引脚`PA1`
- `ADC_CHANNEL_16`对应内部温度传感器

通过设置ADC的通道选择和顺序，可以采集多个模拟信号。

```
// 配置ADC转换通道
ADC_ChannelConfig(ADC1, ADC_CHANNEL_0, ADC_SampleTime_15Cycles);
```

10.2.3 ADC中断和DMA请求

STM32的ADC模块支持中断和DMA请求，以便在ADC转换完成后进行处理。

1. **ADC中断**：ADC可以在转换完成后触发中断，应用程序可以通过中断服务程序（ISR）来处理数据，避免轮询。

   配置ADC中断：

   ```
   // 启用ADC中断
   HAL_NVIC_EnableIRQ(ADC_IRQn);
   ```

   中断服务程序：

```
void ADC_IRQHandler(void)
{
    if (__HAL_ADC_GET_FLAG(&hadc1, ADC_FLAG_EOC)) {
        // 转换完成，读取数据
        uint32_t adcValue = HAL_ADC_GetValue(&hadc1);
    }
}
```

2. **ADC DMA请求**：ADC还支持直接存储器访问（DMA），用于将ADC转换结果直接传输到内存中，而无需通过CPU进行处理。DMA可以大大减少CPU负担，适合高速连续数据采集应用。

   配置ADC使用DMA：

   ```
   HAL_ADC_Start_DMA(&hadc1, adcBuffer, 10);  // 将结果存储到数组adcBuffer中
   ```

   DMA中断服务程序：

   ```
   void DMA1_Stream1_IRQHandler(void)
   {
       if (__HAL_DMA_GET_FLAG(&hdma_adc1, DMA_FLAG_TCIF1_5)) {
           // DMA传输完成，处理数据
           HAL_DMA_CLEAR_FLAG(&hdma_adc1, DMA_FLAG_TCIF1_5);
       }
   }
   ```


10.2.4 ADC转换时间

STM32的ADC转换时间由以下几个因素决定：

- **采样时间**：每个通道的采样时间，可以通过设置不同的采样周期来调节，常见的采样时间为1.5、7.5、15、28等多个选项。
- **分辨率**：ADC的分辨率（例如12位、10位）决定了每个采样周期内需要多少时间来完成转换。较高的分辨率意味着更长的转换时间。
- **时钟频率**：ADC时钟频率越高，转换速度越快。STM32的ADC时钟通常是系统时钟或外部时钟的一个分频。

总的来说，转换时间是由ADC的分辨率、采样时间和ADC时钟频率共同决定的。

```
// 设置ADC通道的采样时间
ADC_ChannelConfig(ADC1, ADC_CHANNEL_0, ADC_SampleTime_15Cycles);
```

10.2.5 ADC数据对齐

STM32的ADC模块支持两种数据对齐方式：**右对齐**和**左对齐**。

- **右对齐**：ADC转换结果的低位被丢弃，数据存储在寄存器的低位部分。例如，12位ADC的结果会存储在数据寄存器的最低12位中。
- **左对齐**：ADC转换结果的高位被丢弃，数据存储在寄存器的高位部分。

左对齐可以使得读取数据时直接获取最高有效位。

选择数据对齐方式：

```
// 右对齐
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
HAL_ADC_Init(&hadc1);

// 左对齐
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Left;
HAL_ADC_Init(&hadc1);
```

10.2.6 ADC的转换模式

STM32支持多种ADC转换模式，适应不同的应用需求：

1. **单次转换模式（Single Conversion Mode）**：每次启动一个转换，转换完成后停止。
2. **连续转换模式（Continuous Conversion Mode）**：在转换完成后自动启动下一个转换，适合实时数据采集应用。
3. **扫描模式（Scan Conversion Mode）**：在多个通道之间依次切换并进行转换，适合多通道采集。
4. **触发模式（Triggered Conversion Mode）**：通过外部信号或内部定时器触发ADC转换。

设置转换模式：

```
// 设置单次转换模式
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;
HAL_ADC_Init(&hadc1);

// 设置连续转换模式
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
HAL_ADC_Init(&hadc1);
```

10.2.7 ADC校准

为了提高ADC的精度和准确性，STM32提供了内置的校准功能。校准通常包括温度传感器和内部参考电压的校准。

1. **ADC校准过程**：在启动ADC之前，可以进行一次或多次校准，以提高测量精度。STM32通常在启动时进行内置的自校准，尤其是对于高精度应用。
2. **校准寄存器**：STM32通过特定的寄存器来进行自校准。校准操作可以通过标准库函数或HAL库来触发。

```
// 启动ADC校准
HAL_ADCEx_Calibration_Start(&hadc1, ADC_CALIB_OFFSET);
```

总结

- **引脚选择和通道配置**：STM32的ADC模块支持多个模拟输入引脚和通道，灵活的通道选择和配置可以实现多种应用。
- **中断和DMA**：通过中断和DMA功能，可以实现高效的数据处理和存储，减少CPU的负担。
- **转换时间和精度**：通过配置采样时间、分辨率和时钟频率，可以调节ADC的转换时间和精度。
- **数据对齐和转换模式**：STM32支持不同的数据对齐方式和转换模式，可以根据应用需求选择最适合的模式。
- **校准**：ADC的内建校准功能可以提高精度，确保转换结果的准确性。

10.3 ADC标准外设库接口函数及应用

在STM32中，ADC模块的配置和操作可以通过标准外设库（Standard Peripheral Library）来实现。标准外设库提供了一系列的接口函数，使得ADC的配置、启动和读取过程更加简便。通过这些函数，用户可以高效地进行ADC采样、转换和数据处理。

10.3.1 ADC标准外设库接口函数

标准外设库为ADC模块提供了丰富的接口函数。以下是常用的ADC相关函数及其作用：

1. **ADC初始化**
   - `ADC_Init()`
   - 该函数用于初始化ADC模块的配置，包括分辨率、数据对齐方式、扫描模式、连续转换模式等。

```
void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct);
```

   该函数的参数：
   - `ADCx`：指定ADC的实例（如`ADC1`、`ADC2`）。
   - `ADC_InitStruct`：包含ADC配置参数的结构体。

2. **ADC通道配置**
   - `ADC_RegularChannelConfig()`
   - 该函数配置ADC的常规通道和采样时间，通常在使用多通道扫描模式时调用。

```
void ADC_RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t
ADC_SampleTime);
```

   参数说明：
   - `ADCx`：指定ADC的实例。
   - `ADC_Channel`：指定要配置的通道。
   - `Rank`：指定通道的排名。
   - `ADC_SampleTime`：指定采样时间。

3. **ADC转换启动**
   - `ADC_SoftwareStartConv()`
   - 启动ADC的转换操作，适用于单次或连续转换模式。

```
void ADC_SoftwareStartConv(ADC_TypeDef* ADCx);
```

   该函数启动ADC的常规转换。

4. **读取ADC值**
   - `ADC_GetConversionValue()`
   - 获取ADC转换后的结果。

```
uint16_t ADC_GetConversionValue(ADC_TypeDef* ADCx);
```

   返回值：ADC转换的结果，返回一个16位值。

5. **ADC校准**
   - `ADC_StartCalibration()`

- 启动ADC的校准过程。校准可以提高ADC的精度，尤其是温度传感器和参考电压的测量精度。

```
void ADC_StartCalibration(ADC_TypeDef* ADCx);
```

该函数启动ADC的自校准过程。

6. **ADC中断配置**
   - `ADC_ITConfig()`
   - 启用或禁用ADC中断，用于在转换完成后触发中断。

```
void ADC_ITConfig(ADC_TypeDef* ADCx, uint16_t ADC_IT, FunctionalState NewState);
```

该函数的参数：

   - `ADCx`：指定ADC的实例。
   - `ADC_IT`：指定ADC中断类型。
   - `NewState`：启用或禁用中断。

7. **ADC DMA配置**
   - `ADC_DMACmd()`
   - 启用或禁用DMA请求。使用DMA可以将ADC的转换结果直接传输到内存中，减少CPU的负担。

```
void ADC_DMACmd(ADC_TypeDef* ADCx, FunctionalState NewState);
```

该函数的参数：

   - `ADCx`：指定ADC的实例。
   - `NewState`：启用或禁用DMA请求。

10.3.2 ADC标准外设库应用编程步骤

使用ADC进行数据采集时，应用程序通常会遵循以下步骤：

1. **ADC外设时钟使能**
   - 在使用ADC之前，必须首先使能ADC外设的时钟。

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);
```

2. **GPIO配置为模拟输入**
   - 配置ADC输入引脚为模拟模式，禁用输入的上拉/下拉电阻。

```
GPIO_InitTypeDef GPIO_InitStruct = {0};
GPIO_InitStruct.Pin = GPIO_PIN_0; // 例如PA0
GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

3. **配置ADC**
   - 使用`ADC_Init()`函数初始化ADC参数（分辨率、数据对齐、转换模式等）。

```
ADC_InitTypeDef ADC_InitStruct = {0};
ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;
ADC_InitStruct.ADC_ScanConvMode = DISABLE;  // 单通道模式
ADC_InitStruct.ADC_ContinuousConvMode = DISABLE;
ADC_InitStruct.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStruct.ADC_NbrOfConversion = 1;  // 仅一个通道
ADC_Init(ADC1, &ADC_InitStruct);
```

4. **配置ADC通道**
   - 使用 `ADC_RegularChannelConfig()` 函数配置ADC通道和采样时间。

```
ADC_RegularChannelConfig(ADC1, ADC_CHANNEL_0, 1, ADC_SampleTime_15Cycles);
```

5. **启动ADC转换**
   - 启动ADC的转换过程。可以使用软件触发启动ADC，或者使用外部触发源。

```
ADC_SoftwareStartConv(ADC1);
```

6. **等待转换完成**
   - 在单次转换模式下，需要等待转换完成后读取转换结果。

```
while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);
```

7. **读取ADC转换结果**
   - 使用 `ADC_GetConversionValue()` 函数获取ADC转换的结果。

```
uint16_t adc_value = ADC_GetConversionValue(ADC1);
```

8. **关闭ADC**
   - 转换完成后，可以关闭ADC以节省功耗。

```
ADC_Cmd(ADC1, DISABLE);
```

10.3.3 ADC标准外设库应用实例

以下是一个完整的ADC应用实例，演示如何使用STM32的标准外设库采集模拟信号：

```
#include "stm32f4xx.h"

void ADC1_Config(void) {
    // 1. 使能ADC1外设时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

    // 2. 配置PA0为模拟输入
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    GPIO_InitStruct.Pin = GPIO_PIN_0;
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
```

```
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

    // 3. 配置ADC
    ADC_InitTypeDef ADC_InitStruct = {0};
    ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;
    ADC_InitStruct.ADC_ScanConvMode = DISABLE;  // 单通道模式
    ADC_InitStruct.ADC_ContinuousConvMode = DISABLE;
    ADC_InitStruct.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStruct.ADC_NbrOfConversion = 1;  // 仅一个通道
    ADC_Init(ADC1, &ADC_InitStruct);

    // 4. 配置ADC通道0，采样时间15个周期
    ADC_RegularChannelConfig(ADC1, ADC_CHANNEL_0, 1, ADC_SampleTime_15Cycles);

    // 5. 启动ADC转换
    ADC_SoftwareStartConv(ADC1);
}

uint16_t Read_ADC_Value(void) {
    // 6. 等待转换完成
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);

    // 7. 获取ADC转换值
    return ADC_GetConversionValue(ADC1);
}

int main(void) {
    // 配置ADC
    ADC1_Config();

    // 读取ADC值
    uint16_t adc_value = Read_ADC_Value();

    // ADC值可以用于控制或者显示等操作
    while (1) {
        // 无限循环
    }
}
```


总结

- **ADC标准外设库接口函数**：提供了配置、启动、读取ADC转换结果等常用操作函数。
- **编程步骤**：包括使能时钟、配置引脚、初始化ADC、启动转换、等待完成、读取结果等步骤。
- **应用实例**：通过配置ADC引脚和通道、启动转换并读取结果，可以轻松实现模拟信号采集功能。